Etherlisp: Implementation and experimentation with a distributed system

Vamvasakis, Andreas M.

Award date:
1993

Awarding institution:
University of Bath

ETHERLISP
Implementation and Experimentation with a Distributed System

submitted by

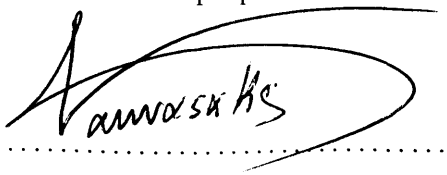
Andreas M. Vamvasakis

for the degree of Ph.D

of the

University of Bath

1993



Signature of Author .....

Andreas M. Vamvasakis

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

Computer networks have been developed 20 years ago or so. Early networks provided elementary networking facilities such as file transfer and simple remote services. In the 1980s the appearance of very high bandwidth Local Area Networks (LAN's) providing standard and highly reliable communication protocols along with cheap and powerful PC's and workstations yielded inexpensive multiprocessor configurations which are an interesting alternative to shared-memory multiprocessors. In this dissertation we investigate the capabilities as well as several critical issues related with network-based systems. In general, this research consists of three logically independent parts. The first one copes with the design and implementation of a network-based system, whilst the second part refers to the appropriate algorithm design for achieving efficient parallelism. The third part deals with the flexibility attained from a network-based system in the sense of numerous ways of expressing and extracting parallelism.

There are also three appendices that complete this thesis. Appendix A stands as a user guide where all embedded primitives are explained. This appendix must be read together with the first chapter. The second appendix illustrates the code solving the classic dinning philosophers problem. Finally, appendix C introduces a naive concurrent approach to the C language.

Acknowledgements

I would like to thank my supervisor Prof. John P. Fitch for his encouragement and guidance during this research. Many thanks to Spyros Kalogeropoulos, Nuong Quang Dinh, and David Lavender who made me familiar with fundamental issues related with this research. I am also graceful to Julian Padget, Russell Bradford, Jeremy Bennett, Michael Dewar, Pete Broadbery, Icarus Sparry, Nigel Phelan, and to all those who eagerly answered to all (and many) of my questions.

Finally, I wish to thank Geoff Smith and Angela Cobban who stand by me in many difficult and tragic moments I had.

Dedications

To the memory of my father

Preface

Computer networks have been developed 20 years ago or so. Early networks provided elementary networking facilities such as file transfer and simple remote services. In the 1980s the appearance of very high bandwidth Local Area Networks (LAN's) providing standard and highly reliable communication protocols along with cheap and powerful PC's and workstations yielded inexpensive multiprocessor configurations which are an interesting alternative to shared-memory multiprocessors. In this dissertation we investigate the capabilities as well as several critical issues related with network-based systems. In general, this research consists of three logically independent parts. The first one copes with the design and implementation of a network-based system, whilst the second part refers to the appropriate algorithm design for achieving efficient parallelism. The third part deals with the flexibility attained from a network-based system in the sense of numerous ways of expressing and extracting parallelism. More precisely, the structure of this thesis consists of several chapters whose major aspects include the following:

The first chapter stands as a brief introduction to the network technologies such as the Ethernet and the Ring packet-switched networks. The chapter continues with a reference of the software required for accessing and controlling the physical network links. This software consists of several communication protocols on top of which any network-based system should be built. This survey makes clear that the underlying structure of a physically distributed system is complicated, inclined in numerous failures, and expensive in resources. As a consequence various questions in regard with design issues, reliability, cost, and performance arise. The chapter concludes with a detailed listing of the objectives which through out this thesis answer to the questions outlined.

Chapter 2 presents the fundamental issues in designing and implementing a simple concurrent network-based system called ETHERLISP. The backbone of ETHERLISP is the standard COMMON LISP enchanced with the appropriate primitives providing concurrency via message-interacting physically independent threads of control. The system's structure is simple but complete since our primary goal is the minimum dependency with the existing Lisp primitives and the achievement of the highest possible parallelism and flexibility.

The third chapter presents the second stub of the ETHERLISP's kernel called FILOS which deals with an efficient transportation of messages across network. The communication cost is the barrier that restricts the performance of network-based systems since the transmission

times are measured in milliseconds. A series of experiments showed that the transmission time is analogous to the byte-size of a message being transported. Based on this fact FILOS provides a method for efficaciously compressing the data quantities enclosed in messages. Apart from compressing messages FILOS is also responsible for a rapid encoding/decoding and multiplexing/demultiplexing of data structures among multiple processes.

In chapter 4 the capabilities of ETHERLISP and of a network-based system in general are explored. A series of experiments are performed and the results are discussed. In addition alternative ways of implementing highly efficient and scalable algorithms are presented.

In the fifth chapter ETHERLISP is used as an environment in which various concurrent paradigms such as Remote Procedure Call (RPC) and LINDA are developed. This survey aims in proving if a network-based system can provide the flexibility and generality required to allow a user to perform extended experimentation with alternative manners of expressing parallelism. This chapter also presents the SOCKET package that is a special set of primitives permitting the manipulation of the raw networking facilities. These primitives along with ETHERLISP yield a more generic environment where single applications can be shared among unrelated users.

Chapter 6 can be thought as an extension of the previous chapter since it introduces a new concurrent paradigm called PRAXIS. Its semantics are novel, few, simple, and powerful enough to provide compact, elegant, and in some cases radical solutions to a broad range of problems. The paradigm provides both synchronous and asynchronous approaches in extracting parallelism with the former case as the most flavoured.

The last chapter mentions several areas of further research where ETHERLISP, FILOS and PRAXIS can or could be improved.

There are also three appendices that complete this thesis. Appendix A stands as a user guide where all embedded primitives are explained. This appendix must be read together with the first chapter. The second appendix illustrates the code solving the classic dinning philosophers problem in the SOCKET package. Finally, appendix C introduces a naive concurrent approach to the C language.

Contents

1 Concurrency via Network-Based Systems 1

1.1 Introduction 1

1.2 Underlying Network Technologies 2

1.2.1 Ethernet: Bus Network Technology 2

1.2.2 ProNET-10: Ring Network Technology 4

1.3 The Internet Protocol (IP) 5

1.4 Network Protocols 7

1.4.1 The User Datagram Protocol (UDP/IP) 8

1.4.2 The Transmission Control Protocol (TCP/IP) 9

1.5 The Socket User Interface 11

1.6 Distributed Programming 12

1.7 The Research Objectives 13

2 The Design and Implementation of ETHERLISP 15

2.1 Introduction 15

2.2 Objectives 16

2.3 The ETHER Kernel 16

2.3.1 Creating Remote Threads of Control 18

2.4 Messages: Types and Properties 19

2.4.1 Constructing Messages 20

2.5 Message Passing 21

2.6 Message Operations 23

2.7 Initializing a Distributed Application Program 24

2.8 Object Properties in a Distributed Address Space 25

2.9 Global Error Handling 25

2.10 Process Termination 26

2.11 Examples 26
2.11.1 On Establishing a Remote Login Session 27
2.11.2 Selective Remote Bindings 27
2.11.3 Scheduling and Synchronizing Distributed Processes 28
2.11.4 Synchronization via Monitors 29
2.12 Summary 30

3 On Minimizing the Network Overhead **31**
3.1 Introduction 31
3.2 Objectives 32
3.3 Analysis of the Network Overhead 32
3.4 Issues on Compression of LISP Object Structures 34
3.5 Logical Message Compression 36
3.5.1 The System-Index Table 36
3.5.2 The Hash-Index Table 37
3.5.3 The Atom-Index Table 37
3.5.4 A Graphical Presentation of the Index Tables. 38
3.5.5 Miscellaneous Issues on Index Tables 39
3.6 Physical Message Compression 40
3.7 Variable Length Object and Index Blocks 41
3.7.1 New-object Blocks 42
3.7.2 Index Blocks 43
3.7.3 The Low-level FILOS Compression Format 44
3.8 The Encoding/Decoding Algorithm 46
3.9 Compression Performance 46
3.9.1 Special Cases of Compression Performance 48
3.9.2 Compression Results 48
3.9.3 Behaviour of the Index Tables 50
3.9.4 Persistent Compression via History Indices 51
3.10 Speed Performance 51
3.11 Summary 54

4 Exploring the Capabilities of Network-Based Systems **55**
4.1 Introduction 55

4.2 Objectives 56
4.3 Issues on the Characteristics of Distributed Algorithms 56
4.3.1 Notions on Measuring the Performance of Parallel Algorithms 58
4.4 Case Study I: Matrix Multiplication 59
4.4.1 Matrix-by-Matrix Multiplication 59
4.4.2 Timing Results 61
4.4.3 Vector-by-Matrix Multiplication 62
4.5 Case Study II: Searching 62
4.5.1 Timing Results 65
4.6 Case Study III: Sorting 65
4.6.1 Timing Results 67
4.6.2 Extracting Parallelism when Data Dependency is Loose 69
4.6.3 FILOS's Effects on the Performance of Parallel Algorithms 70
4.7 Discussion 71
4.8 Summary 74

5 Building Autonomous Abstract Systems on Top of ETHERLISP 75

5.1 Introduction 75
5.2 Objectives 76
5.3 Implementing Remote Procedure Calls (RPC) 76
5.4 CMU Common Lisp 80
5.5 Avalon/Common Lisp 82
5.6 Implementing LINDA 84
5.6.1 On Applying FILOS on LINDA Tuples 87
5.7 The EtherLISP's SOCKET Package 88
5.7.1 Reliable Communication Service (RCS) 89
5.7.2 Connectionless Communication Service (CCS) 90
5.7.3 Socket Address Service (SAS) 90
5.7.4 EtherLISP+ and Sun/RPC 92
5.7.5 XDR and FILOS 93
5.8 Summary 94

6 Implementing Innovative Techniques on Top of ETHERLISP 95

6.1 Introduction 95

6.2 Batch Message Passing (BMP) 96
6.3 PRAXIS: A Paradigm for Parallel Execution of Instances 99
6.3.1 Synchronizing Processes in PRAXIS 101
6.3.2 The Asynchronous Approach 103
6.3.3 Performance Measurements 105
6.3.4 Simplicity and Expressiveness 109
6.3.5 PRAXIS versus LINDA and TIME WARP 110
6.3.6 Synchronizing Simultaneous Access on Shared Data 114
6.4 Summary 119

7 Further Research and Concluding Remarks **120**
7.1 Introduction 120
7.2 Extended Process Interconnections in ETHERLISP. 120
7.3 Enchancing FILOS 122
7.4 PRAXIS in a Shared-Memory Environment 123
7.5 Conclusions 125

A ETHERLISP User's Manual **134**
A.1 Introduction 134
A.2 The Fundamental Primitives 135
A.3 Special Variables 136
A.4 Message Manipulation Primitives 137
A.5 General Purpose Primitives 138

B The Dining Philosophers Solved in the SOCKET Package **141**
B.1 Defining the Problem 141
B.2 Solving the Problem 142

C EtherC: A Concurrent Approach to the ANSI C Language **147**

List of Figures

1-1 The format of a frame as it travels across an Ethernet. 4
1-2 The ProNET-10 frame format. 4
1-3 Interconnection of incompatible networks by gateways. 5
1-4 Format of an Internet datagram. 6
1-5 Demultiplexing network frames. 8
1-6 Format of a UDP datagram. 9
1-7 Format of a TCP segment. 10
1-8 The generic layer hierarchy of a network-based distributed system. 12

2-1 The architecture of ETHERLISP. 17
2-2 Mutable and non-mutable object handling. 26
2-3 Code for the *rlogin* Unix command. 27
2-4 Selective remote bindings. 28
2-5 Load-based criteria for selecting readable communication channels. 29
2-6 Construction of a monolithic monitor. 30

3-1 A Graphical Presentation of the Index Tables. 39
3-2 The generic structure of variable size object blocks. 43
3-3 The low-level compression output format. 45
3-4 The format of an example message. 47
3-5 LISP program's compression and timing diagrams. 52

4-1 Multiple-Instruction, Multiple-Data (MIMD) processor architecture. 57
4-2 Matrix-by-matrix multiplication: The sequential algorithm. 60
4-3 Matrix-by-matrix multiplication: The parallel algorithm I. 60
4-4 Matrix-by-matrix multiplication: The parallel algorithm II. 61
4-5 Searching: The sequential algorithm. 64

4-6 Searching: The parallel algorithm. 64
4-7 Quicksort: The sequential algorithm. 67
4-8 Quicksort: The parallel algorithm. 68
4-9 Graphical representation of the case studies. 72

5-1 The components of Birrell and Nelson's RPC system and their interactions for a simple remote procedure call. 77
5-2 Code of an abstract RPC interface on top of ETHERLISP. 80
5-3 The basic constructs of Avalon/Common Lisp on top of ETHERLISP. . . . 82
5-4 Code for the basic LINDA primitives when operating on a single global TS. 85
5-5 An example LINDA program developed in ETHERLINDA. 86
5-6 Reliable Communication Service (RCS): The Basic Procedure. 91
5-7 Connectionless Communication Service (CCS): The Basic Procedure. 91
5-8 Socket Address Service (SAS): The Basic Procedure 91
5-9 The structure of a generic distributed application in ETHERLISP+. 93

6-1 Scheduling processes via Batch Message Passing (BMP). 97
6-2 Demonstrating the Batch Message Passing (BMP) concept. 98
6-3 Procedure-Invocation Condition-Synchronization Message Passing (PICS/MP).100
6-4 Matrix-by-matrix multiplication: An asynchronous solution in PRAXIS. . . . 104
6-5 The Left-to-Right-Urgency (LRU) scheduling policy of PRAXIS. 115
6-6 The dining philosophers solved in ETHERLINDA and PRAXIS. 117
6-7 A prominent solution of the dining philosophers problem in PRAXIS. 118

7-1 Alternative schemes for inter-connecting memory-disjoint processes. 121
7-2 Handling of tuple-like messages. 123
7-3 Sorting in PRAXIS. 124

B-1 The dining philosophers solved in the SOCKET package. 146

C-1 Th echo server. 148
C-2 The echo client. 149

List of Tables

3.1 Compression results: Sizes are measured in bytes, and the percentages refer to the data quantity excluded. 49

3.2 Behaviour of compression indices. 50

3.3 FILOS' speed performance (in milliseconds). 53

4.1 Matrix-by-matrix multiplication: Comparative results between algorithms I and II when $P = 2$, speedup expressed in percentages (%), $S^{\star}$ the additional speedup in % of algorithm II over I, S^{II} and E^{II} II's speedup and efficiency, and time measured in seconds. 63

4.2 Vector-by-matrix multiplication: Comparative results between algorithms I and II when $P = 2$, speedup expressed in percentages (%), $S^{\star}$ the additional speedup in % of algorithm II over I, S^{II} and E^{II} II's speedup and efficiency, and time measured in seconds. 63

4.3 Searching $1, \frac{n}{2}, n$-length random numerals in a n-length random sequence when $P = 1, 2, ..., 6$, speedup is expressed in percentages, and time is measured in seconds. 66

4.4 Quicksorting integers when $P = 1, 2, .., 10$, speedup is expressed in percentages, and time is rounded up to one decimal and measured in seconds. . . . 69

4.5 Sorting independent numeral sequences, when $P = 1, 2, ..., 4$ and time is measured in seconds. 70

4.6 Comparative results when quicksorting n one-digit integers (I) and n symbols and strings (II) when $P = 2$, and time measured in seconds. 70

4.7 Comparing the critical features of distributed algorithms when, t_{dist} is the data distribution cost, t_{exec} is the execution time of a subinstance, t_{corr} is the data correlation cost, t_{udp} is the UDP/IP cost, and P is the number of PE's. 73

4.8 Timing TCP/IP and UDP/IP transmission speeds (time in milliseconds). . 73

4.9 Comparative results between TCP/IP and UDP/IP endpoints when quick-sorting integers, $P = 4$, $D_{\overline{x}} = \sum_{i=0}^{n-1} \mid \overline{X} - x_i \mid$, and time measured in seconds. 74

5.1 Compression performance on LINDA tuples. 87

6.1 Matrix-by-matrix multiplication: Comparative results between PRAXIS, LISP, and ETHERLISP (time measured in seconds). 107

6.2 Matrix-by-matrix multiplication: Comparative results between LISP, ETHERLISP, and Synchronous PRAXIS (time measured in seconds). 108

6.3 Measuring PRAXIS performance when multiple co-processors are emploied. . 108

Chapter 1

Concurrency via Network-Based Systems

1.1 Introduction

IT IS BECOMING APPARENT that future requirements for processing speed, system reliability and cost-effectiveness will result in the development of alternative hardware architectures, as well as in software schemes. The concept of *concurrency* has been considered as the major way to cope with these goals. At the hardware level, concurrency refers to the existence of multiple processing units executing machine instructions in parallel. Similarly, concurrent programs specify two or more processes that cooperate in performing a task, where a *process* can be thought of as a single line of control in a program. Currently concurrent hardware systems have been roughly classified [Flynn72] either as *SIMD* (single-instruction-stream, multiple-data-stream), or *MIMD* (multiple-instruction-stream, multiple-data-stream). The first hardware class is suitable for applying a common operation, or a set of operations, to many separate sets of data, whereas the second class is suitable for applying different operations, or sets of operations, to separate (or even common) sets of data.

Technology has evolved a variety of *multiprocessor* systems; the most popular architectures

are the *shared-memory* multiprocessors where multiple processing units share a common memory, and the *network-based* systems where numerous single or multiprocessor processing units (nodes) share a communication network. The rapid advances in the development of effective communication networks yields efficient network configurations that are a very important alternative to the shared-memory configurations.
A brief introduction to the underlying network technology, [Comer88, SKPW89], is essential for two reasons. First, the comprehension of the basic network construction makes the reader familiar with the terminology referred in the next chapters, and second, the weak points of network-based systems become more apparent.

1.2 Underlying Network Technologies

There are networks of various technologies, sizes and topologies serving the needs of individual users. This section presents the fundamental packet-switching concepts and technologies for consolidating the so called *physical layer* used for transferring information among geographically separated sites. In general, communication networks can be classified into two categories:

- **Circuit-switched networks**, where the connection established (circuit) between two network nodes is dedicated. The telephone network is an example to this case. Circuit switching is of guaranteed bidirectional traffic but its cost is high depending on the duration of the call.

- **Packet-switched networks**, where the traffic is divided into small segments (packets) containing the destination address along with few bytes of data. The later implies that packet switching allows cheap concurrent multiple communication among nodes. On the contrary, the increase of traffic yields to a poor network capacity. However, packet switching is mainly used for connecting computers sharing a network.

Both network technologies can also be divided into *long haul networks* and *local area networks*. The chief difference lies in speed where long haul networks' speed ranges from 9.6Kbps to 1.5Mbps, whilst local area network operate between 3Mbps and 100Mbps.

1.2.1 Ethernet: Bus Network Technology

Ethernet is a computer network invented at Xerox PARC in the early 1970s, and its major architectural principles include a shared multiaccess bus with distributed control. The basic

component of an Ethernet is a coaxial cable, called the *ether*, of about half inch in diameter and up to 500 meters in length. The ether is a completely passive wire whilst the active Ethernet's components are attached on the connected computers. Despite its short span, an Ethernet can be expanded with devices called *repeaters* that relay signals from one ether to another. Individual computers, or stations, connect to the ether by *taps* allowing small pins to touch the backbone wire through holes in the outer layers of the coaxial cable. A so called *transceiver* is connected at the tap for sensing and sending signals on the ether. The signals reach the actual receiver (station) via a *host interface* located on the station. This architecture permits the easy reconfiguration of stations, for example moving machines from one point to another without taking down the network.

As it is mentioned earlier, the basic properties of an Ethernet are the *bus* technology, i.e. the sharing of a single cable among all attached stations, the *broadcast* facility, i.e. all stations receive every transmission, and finally, *distributed access control* since there is no central control authority. This set of mechanisms make the Ethernet known as *carrier sense multiple access with collision detection (CSMA/CD)*. When a station wishes to send a packet the ether must be idle, otherwise a carrier sense mechanism forces the station to defer if any transmission is in progress. Since signals travel at a (theoretical) rate of 10Mbits per second (approximately 80% of the speed of light [Comer88, p:16]), it is possible for two or more stations to start transmitting simultaneously; the later has as an effect the scrambling of the crossed signals. This phenomenon is called *collision*. When a collision is detected the station interrupts transmission and retransmits after a random period of time determined by a binary exponential backoff algorithm. However, experiments reveal that under normal load the radio-based **Aloha** packet switching network [JSJH80] completes the 99.18% of the transmissions with zero latency, 0.79% are delayed due to defence, and less than 0.03% of the packets are involved in collisions.

The transceivers pass all packets onto the host interface which filters them by means of accepting only packets for the specified station. This is achieved by an *addressing mechanism* where each interface has a 48-bit ethernet address. The address is an inseparable part of an interface and hence, if the interface moves to another machine the address moves with it. Packet addresses and data along with other information travel enclosed into frames of variable length. More precisely, an ethernet frame, figure 1-1, is *self-identifying* because it contains the addresses of the sender and the receiver. The type field denotes the protocol software (section 1.4) that handles packets; thus, multiple protocols can be used on a single

Preamble	*Destination Address*	*Source Address*	*Packet Type*	*Data*	*CRC*
64 bits	48 bits	48 bits	16 bits	368-12000 bits	32 bits

Figure 1-1: The format of a frame as it travels across an Ethernet.

station without interference. One of the major objectives of Ethernet is to provide reliable communication. This is guaranteed by the Cyclic Redundancy Check (CRC) field; a packet upon receipt must contain the same CRC value as it was computed by the sender.

1.2.2 ProNET-10: Ring Network Technology

Another popular network technology is the ring configurations named as *token-passing.* A commercial network of this type, which is an important alternative to the Ethernet, is the ProNET-10 manufactured by Proteon Inc. The basic characteristics include a 10Mbps capacity, short geographic span, and an active host interface for each attached station. The physical link is not a continuous wire but it consists of point-to-point connections among the interfaces. More precisely, an interface has two lines for sending/receiving frames to/from the neighbouring stations. A frame travels around the ring entering the receive line and exiting from the send line of each interface until it reaches its initial sender. At this point, the ring is terminated and the frame is prevented for looping for ever. The term token-passing refers to the existence of a token that continually circulates the ring. Any station wishing to transmit must remove the token from the ring. When the frame completes a circle, the token is released allowing other stations, in a non deterministic way, to use it. Like Ethernet, data are encapsulated within frames, figure 1-2, of variable length. The address length is

Start of msg	*Dest. Addr.*	*Src. Addr.*	*Packet Type*	*Data*	*End of msg*	*Parity*	*Refuse*
10 bits	8 bits	8 bits	24 bits	0-16352 bits	9 bits	1 bit	1 bit

Figure 1-2: The ProNET-10 frame format.

only 8-bit long because ring interfaces do not have fixed addresses assigned by vendors. An authority must ensure that no two nodes on a network have the same address. The

refuse bit is cleared upon receipt of a frame, otherwise the frame is considered undelivered. Notable is the increased data capacity of a ring frame over the ethernet's one, as well as the remarkable fewer frame administration bits. The main drawback of the ring is the loss of the token due to hardware failure (host crash, electrical interference) causing the interruption of the ring. An efficient and reliable mechanism envisages this situation by generating a new token. Under heavy load the number of collisions on an Ethernet rise whilst the bandwidth reduces, fact that is not true for a ring network. On the other hand, when a ring station crashes the whole network comes down. Note that the Cambridge Ring [Needham79, Penny82] is an interesting implementation of the ring network technology.

1.3 The Internet Protocol (IP)

The existence of multiple underlying network technologies arise a problem. The interconnection of numerous disparate physical networks as a co-ordinate unit. The solution of this problem is called *internetworking* which is another technology that joins individual and incompatible networks together into a single virtual network named an *Internet.* Clearly, one ethernet station can communicate with a ring station as if they were nodes of the same physical network.

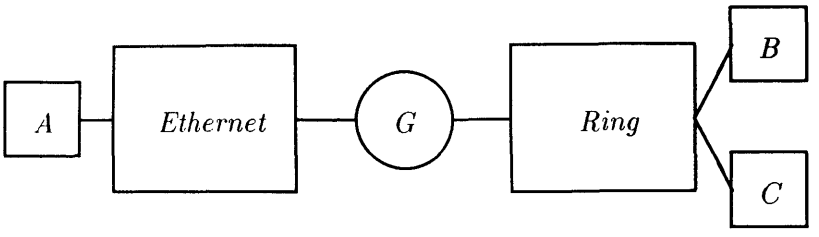


Figure 1-3: Interconnection of incompatible networks by gateways.

Apparently, one may ask the way data packets travel through different networks, and how they are demultiplexed to their final receiver since host addresses differ from network to network. *Gateways* are (usually) special purpose stations that accept packets from one network and forwards them to destination stations, or gateways on another network. In figure 1-3 the ethernet station A communicates with the ring stations B and C via the gateway G. Obviously, packets can travel in the opposite direction.

Internet stations could be identified as a tuple <*name, address, route*>, where *name* indicates what the object is, *address* identifies where the object is, and *route* denotes how the

object can be reached. The Internet *addressing mechanism* is a method that any internet station has its own unique address assigned by a central authority [1]. Because the topology of an Internet at a future time is unknown, internet addresses consist of two portions. The network part corresponds stations to networks, and the local part identifies stations on the specified network. Internet addresses are divided into three classes. *Class A* identifies few networks and many hosts, *Class B* identifies approximately the same number of networks and hosts, and *Class C* identifies many networks and few hosts. [2]

Internet, like every packet switching network, carries its data encapsulated into *Internet Protocol Datagrams.* Transportation of IP datagrams is feasible because they are enclosed into the data portion of underlying network frames, whilst the frame's type field is set to "type IP" . Upon receipt the type field guides the receiver to demultiplex the frame to the software that handles IP datagrams. Note that IP datagrams can travel via any network technology carring frames. The format of an IP datagram, figure 1-4, differ from network

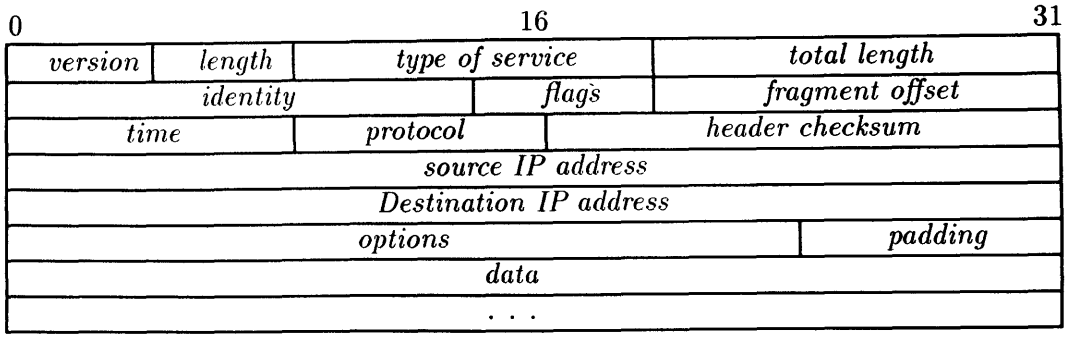


Figure 1-4: Format of an Internet datagram.

frames since the IP software handles many different cases. The most important fields include the *time* field that prevents (orphan) datagrams looping for ever; the ***identification*** field identifies siblings of a segmented datagram, or different datagrams sharing the same source. Datagrams may be segmented into smaller datagrams because they do not fit in a frame of networks of a specified technology. The disadvantage of datagram segmentation is the need for assembling all segments into a complete datagram at the final receiver's site. Consequently, each segment encapsulates a copy of the datagram header (the first six lines of the datagram in figure 1-4). The later also implies that the sender maintains a counter that it increments each time it sends a datagram; thus, every datagram has its

[1] The *Network Information Center (NIC).*

[2] Internet addresses are usually given in a dotted notation (four dot-separated numbers).

unique identification number. Finally, the *type of service* field denotes that datagrams are manipulated either as low or high delay.

A couple of importance cases the Internet must cope with are the *mapping* of internet addresses to physical addresses, and the *routing* of datagrams. In the first case, the Address Resolution Protocol (ARP) is the authority that corresponds software IP addresses to hardware-depended physical addresses. ARP maintains a table with the addresses of all internet stations obtained as the result of an ARP request sent to any host.

The routing problem refers to the topology of the whole internet onto a table called the routing table located on every gateway. Thus, the network portion of an IP address guides a gateway to forward a datagram to the destination gateway, and from there to the destination network. Usually, the routing tables are dynamically updated in order to reflect any change in the internet topology. Internet, like underlying networks, is not absolutely secure in delivering datagrams. The *Internet Control Message Protocol (ICMP)* recovers from erroneous or unsuccessful datagram transmissions; strictly speaking, gateways send ICMP messages when they cannot route a datagram, or instruct a station to use another (alternative) gateway.

1.4 Network Protocols

So far, we have described the underlying hardware network technologies and we have touched lightly upon the Internet software mechanism that, as a co-ordinating unit, carry data among physically distributed network stations. The network software consists of multiple *protocols* that are methods agreed between two nodes for communication. In general, specific protocols are required to cope with communication problems such as:

- **Hardware/software failure**: A protocol must detect and recover host, gateway, or network link failures, as well as operating system crashes.
- **Network congestion**: A guard protocol prevents the exceeding of the finite network capacity.
- **Data corruption**: Factors, like external magnetic fields, may corrupt the travelling electrical signals.
- **Packet loss, delay, or duplication**: Specific protocols are also needed to detect and recover from packet loss, delay, data duplication, and data sequence errors.

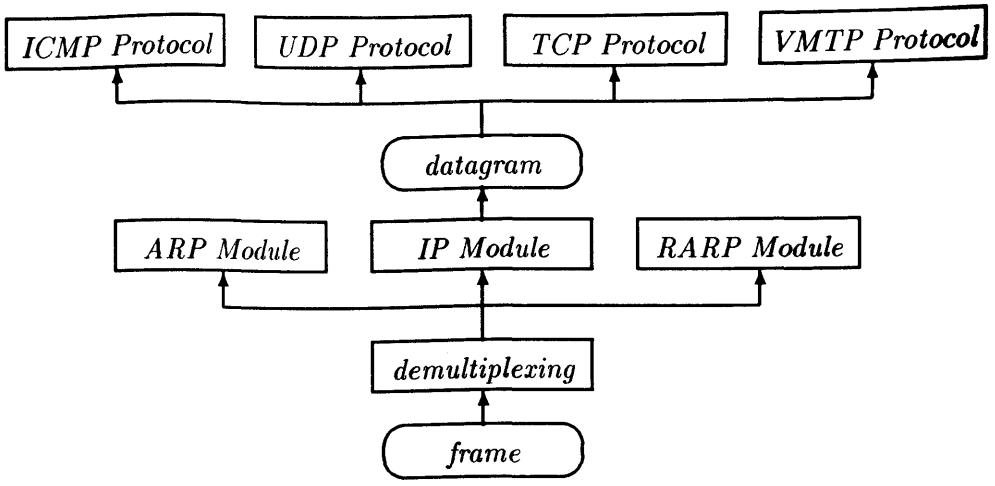


Figure 1-5: Demultiplexing network frames.

The Internet protocol is responsible of routing IP datagrams to the destination station. Usually, stations provide a multiprocessing environment where multiple user processes (applications) can execute concurrently; consequently, datagrams must be somehow *demultiplexed.* Figure 1-5 illustrates the demultiplexing of a network frame, based on the frame type field, among multiple modules. The generated IP datagram is further demultiplexed among numerous protocols which eventually deliver data to the specified receiver. Clearly, the opposite procedure is called *multiplexing.*

Sometimes it is desirable for a sending process to contact a destination process known by its service, or a destination process that has been dynamically replaced. The identification of such destinations is achieved by *protocol ports* which are positive integers. Therefore, the sender must know both the Internet address of the destination station and the port number of the destination on that station. This can be graphically stated in figure 1-5 as a further demultiplexing performed by protocols.

These techniques are some of the major tasks of the high level protocols and the most two important protocols are briefly presented next.

1.4.1 The User Datagram Protocol (UDP/IP)

The *User Datagram Protocol (UDP/IP)* is not a complete protocol and it is built upon the Internet Protocol. It provides the basic mechanisms to carry datagrams among stations. Consequently, UDP/IP is an unreliable and connectionless delivery service where datagrams may be lost, duplicated, or delivered out of order. The absence of acknowledging sent

datagrams may lead to an inability of the receiver to process rapidly incoming **datagrams**. The basic transferred data unit is called a *user datagram*, figure 1-6, which is **encapsulated** into the data portion of an IP datagram. Every datagram is preceded by a **pseudoheader**

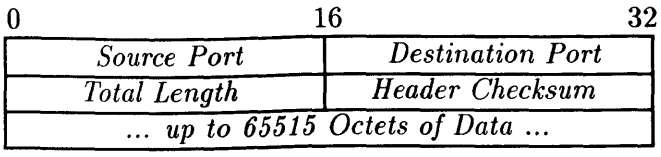


Figure 1-6: Format of a UDP datagram.

containing the IP source and destination addresses, the IP protocol type (UDP/IP in this case), and the the length of the user datagram. Strictly speaking, as a network packet passes from lower to higher layers, a header is removed at each stage; thus, the ultimate destination receives the actual data. Note that the innermost header corresponds to the highest protocol (i.e. UDP/IP), whilst the outermost one corresponds to the lowest layer (i.e. Ethernet). UDP/IP is suitable for applications requiring a datagram-oriented communication, or as a building block for implementing other special purpose protocols.

1.4.2 The Transmission Control Protocol (TCP/IP)

The *Transmission Control Protocol (TCP/IP)* is an independent general purpose protocol that transports, the possibly lost, duplicated, corrupted, or reordered IP datagrams, into a reliable full-duplex stream of characters. Its reliability emerges from the permanent and dedicated connection established between (only) two communicating IP sites. Data flows as a continuous stream of octets, and therefore fragmented into smaller *segments* that fit inside IP datagrams (the data field in figure 1-4). Every sent segment is acknowledged; otherwise it is retransmitted. The unique identification number of each segment ensures the right order in delivery.

TCP/IP belongs in the *sliding window protocols* family based on a *stop-and-wait* method, where any transmission is interrupted until the acknowledgement of the last sent segment. This is one of the major disadvantages of TCP/IP because most of the underlying networks carry data (electrical signals) in both ways at the same time. Note that things could be improved if the window size increases allowing multiple packets to be transmitted within one transmission.

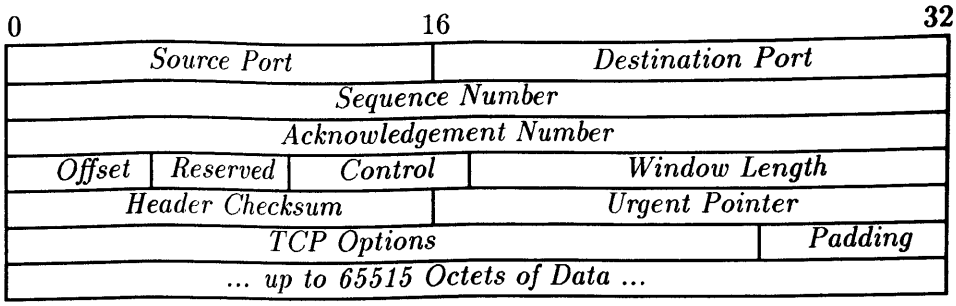


Figure 1-7: Format of a TCP segment.

The format [3] of a TCP/IP segment is illustrated in figure 1-7. Like UDP/IP, TCP/IP encloses in each segment the port numbers of both sender and receiver. The sequence number field designates the position of the first byte in the stream. Upon receipt, the receiver acknowledges the segment by means of sending the last octet received, i.e. an offset depending on the segment's sequence number. Flags contained in the *control* and options fields provide multiple important facilities:

- **Pushing data**: Octets can be buffered before transmission for reasons of efficiency. Clearly, the sending procedure is delayed. However, in some cases any amount of data (either a single octet) require immediate transmission, and a *push* mechanism signals the buffer's evacuation.

- **Establishing connections**: The *SYN* control bit field is used for establishing permanent circuit-like connections.

- **Customizing segment size**: Both the sending and receiving sites negotiate for an appropriate segment size.

- **Terminating connections**: When the *FIN* control bit is set upon receipt of a segment the segment is acknowledged and the connection is closed.

The *urgent* mechanism is another important aspect allowing the delivery of data, such as interrupt signals, called *out-of-band* data as soon as possible. Unlike UDP/IP datagrams, segments are not independent and the unsuccessful delivery even of one of them yields the retransmission of all. A segment needs to be retransmitted if the *round trip time (RTT)*

[3] Each segment is prepended by a pseudoheader containing the IP addresses of both source and destination, the protocol type, and the length of the segment.

expires. The RTT is the elapsed time between the sending of a segment and the receive of an acknowledgement.

TCP/IP also contributes to the smoother data flow on the Internet by either shrinking the window size, or by shrinking the size of the *congestion window.* Congestion is possible when networks of different bandwidth coexist. Thus, slow networks can not process rapidly incoming datagrams and either the datagrams are ignored and hence requiring retransmission, or the gateways are overloaded and the throughput of the whole network decreases. Clearly, the congestion window denotes the amount of data [4] the underlying network can carry.

1.5 The Socket User Interface

The description of the underlying Internet protocols, residing in the machine's operating system, up to this point does not specify any implicit or explicit interface between protocols and user application programs. The basic building block for interprocess communication and networking, first introduced in the 4.2BSD release of Unix [SunOS, Vol:10], is the *socket* abstraction that creates an endpoint for communication. Unlike pipes and socketpairs, sockets provide communication between any two processes that have no common ancestor; furthermore, processes can be located on distinct processors.

The main attribute of the socket mechanism is the resemblance with the *open-close-read-write* I/O operations of the Unix [BKRP84, Leffler89] file access system. A communication channel is considered as a file (or device) identified by a descriptor. Data can be read or written [5] via the socket descriptor when an *open* operation has been performed. Similarly, when the data transfer is complete a *close* operation is required. The chief difference between files and sockets is that files are bound to file descriptors upon an *open* system call, whilst sockets use names to refer to one another each time they exchange information. More precisely, socket names are converted into Internet addresses specified in a space called *domain.* Since multiple protocols may be active on the same host, an Internet address is specified by a triple including a protocol, the port and the host's address; consequently, different protocols may use the same ports. In case of sockets, port numbers can be thought of as the number of a mailbox where protocols places data addressed to a specific socket. Strictly speaking,

[4] TCP/IP always selects the smaller window.

[5] After a TCP/IP connection has been established two processes would communicate via the ordinary Unix functions *read()* and *write()*.

every pair of communicating sockets is specified by a temporary or permanent ***association*** that is a tuple including the chosen protocol, the address of the local host, the local port number, the address of the remote host, and the remote port number. Currently, there are two main types of sockets; the *stream* sockets that interface the TCP/IP protocol, and the *datagram* socket interfacing the UDP/IP protocol.

1.6 Distributed Programming

The advances in network technology provide the development of *distributed systems* that provide access to real and abstract objects or to resources where the distributed nature of the system is usually hidden as much as possible. Real objects are entities such as processors, I/O devices, etc., whereas abstract objects are entities such as files, directories, etc. Abstract objects can be used as basic building blocks for creating higher level objects such as concurrent processes. There are two possible kinds of objects; the *active*, where the distributed system model is of a process which can be thought to be executing on a real or virtual processor, and *passive*, such as I/O devices, communication channels or files. Distributed processes co-operate by communicating for the achievement of a single goal. Communication and synchronization is accomplished by means of *message passing*, where a process sends a message that is received by another.

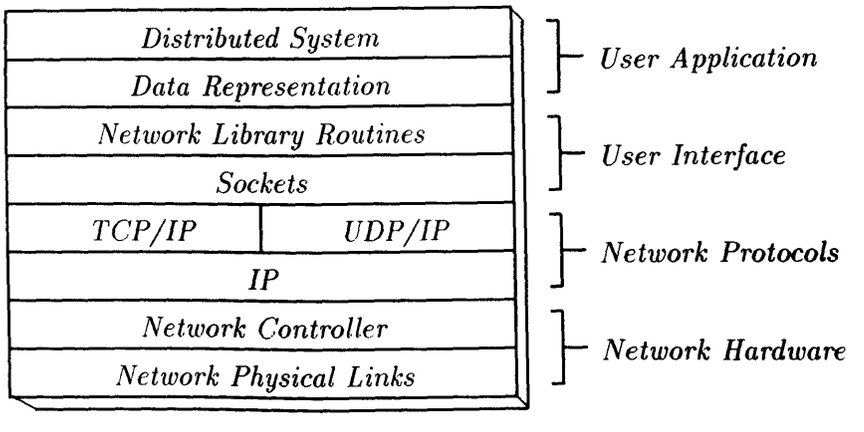


Figure 1-8: The generic layer hierarchy of a network-based distributed system.

Any network-based distributed system implies the utilization of all layers illustrated in figure 1-8. Apart from the layers provided by the operating system, the user must specify a protocol that preserves the semantics of the system when data travel across network. This protocol, or set of rules, is analogous to the *External Data Representation (XDR)* library

[IRIS-4D87], which additionally, accounts for differences in the internal data representation of different computer architectures. Despite the increased performance and transparency of all layers beneath a distributed system, the potential computing capacity would be poor due to numerous reasons. Although some factors depending on the technology, such as the delay and latency of the media yield a strong influence, the design and semantics of a distributed system must be also considered. The development of a network-based distributed system is open to numerous questions:

- What language features are necessary to support distributed processing beyond these required in languages that are used in uniprocessor systems?
- What mechanisms are required to support different data and processing schemes?
- How should both processing and data be distributed?
- How should data be represented and efficiently transmitted?
- How should distributed processes be organized for control and communication?
- What the optimum interprocess communication model is?
- How much should a distributed system be transparent?
- How should distant error and termination control be guaranteed?

1.7 The Research Objectives

The main objective of this research is the design and implementation of a network-based Lisp concurrent system which answers all of the questions outlined above. In particular, our primary goals and attention are drawn to the development of a simple but integrated and efficient system which should provide the following features:

▷ **Simple structure and semantics:** The simplest structure of a standard integrated physically distributed system should provide concurrent processing, interprocess communication, and process synchronization. Based on the reliability of the employed communication protocols and assuming normal network conditions aspects such as persistent communications, process migration, and concurrency within single processes which increase complexity and overhead are not (currently) considered. Meanwhile, we believe that simple and well defined semantics entail the minimum alterations and conflicts and a harmonic co-existence with the host Lisp semantics.

- ▷ **Transparency:** Transparent communication, control, and synchronization mechanisms in the sense of providing a highly abstract parallel machine.

- ▷ **Efficiency:** Provision of an efficient environment yielding processing power which is far in excess of the one achieved from a uniprocessor. We advocate that high efficiency can be attained mainly by two ways: many primitives that perform the absolutely necessary tasks instead of few, generic, and hence complicated ones, and a drastic reduction of the communication cost

- ▷ **Flexibility and scalability:** A flexible concurrent environment that provides alternative ways of solving a broad range of real problems. Effortless construction of highly scalable algorithms that allows the system to extend independently from the algorithms being executed as the demand for additional processing power grows.

- ▷ **Wide experimental environment:** The provision of a wide environment in which extended experimentation with numerous existing concurrent paradigms or new ways of extracting parallelism to be carried out.

- ▷ **Generality and extensibility:** We have in view a general system that allows its extension with other important aspects such as the ones outlined in the first objective. Finally, we aim at the provision of a subsidiary set of primitives that allow low-level operations on the underlying networking mechanisms; thus, the user is provided with unbounded control and power for developing any sort of networking facilities similar to the ones offered by distributed operating systems.

Chapter 2

The Design and Implementation of EtherLISP

2.1 Introduction

THE FIRST STEP OF OUR RESEARCH is the presentation of the fundamental built-in primitives of the ETHER kernel which provides network-based concurrent capabilities to Common Lisp[Steele90]. The host language is the AKCL Common Lisp developed in Kyoto of Japan and enchanced in Austin of Texas; the release 1.530 is supported on Sun's 3/60 running SUNOS 4.2BSD. The ETHER kernel in combination with AKCL yields a concurrent dialect called ETHERLISP. The new language has been designed to preserve the semantics of Lisp, while the embedded primitives co-operate with the original ones in total harmony for the achievement of a single goal.

In this chapter we present a survey of the basic properties of concurrent systems and the way our system copes with them. A series of examples contribute to a deeper understanding of the incorporated semantics. Finally, a detailed description of all built-in primitives is given in appendix A.

2.2 Objectives

The objectives of this chapter are focused on the accomplishment of the goals stated in the next conjecture:

■ **Conjecture 1** *The minimal requirements of a simple but efficient physically distributed (network-based) system, include the existence of distinct processors, a high speed interconnection network, and an interface providing the entire system as a single transparent processing unit.*
Despite of the minor or major differences among physically distributed systems, the following three features must be in common and therefore, to be fulfilled by the simple distributed system.

- *The ability to express concurrent execution, that is execution of co-processes overlap in time.*
- *Interprocess communication, where co-processes executing on distinct physical processors can exchange information.*
- *Process synchronization, where competitive co-processes execute in a mutual exclusive manner.*

2.3 The ETHER Kernel

The fundamental concept of concurrency is that of *process* which generally may be thought of as a program in execution. The main features of a process include its *environment* or *context*, and a set of *operations* or *actions* that occur in that environment, while the environment consists of a set of definitions, called the *address space*, and a set of *resources*. In a multiprogramming environment a process is also determined by another two aspects; the *template* from which the process is created that consists of the code and the context of the process, and the *instance* which is the execution of all or part of its code in its context. As the process' execution proceeds its context is changed, and the so called *state* of a process is determined by its current context and the amount of the code executed so far. Functional programming allows the division of the template of a process into a set of blocks, called routines, procedures, or functions, each performing a specific action. Therefore, each block can be thought of as a sequential process and a program can be considered as a collection of

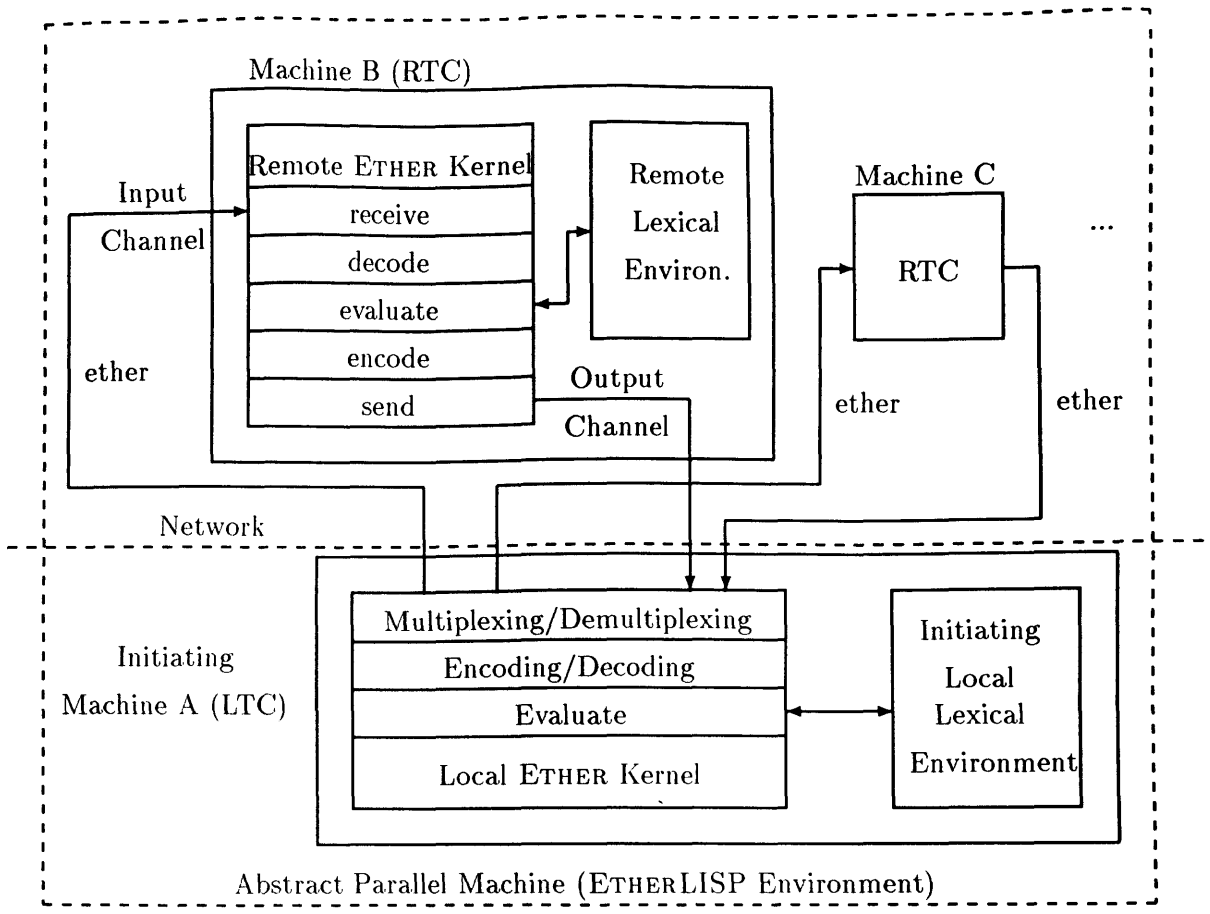


Figure 2-1: The architecture of ETHERLISP.

processes rather than a single process. In some concurrent systems each sequential function can be transformed into an independent thread of control that may execute simultaneously with others. Such systems are called *fine-grained* concurrent systems and they are suitable for shared-memory configurations. Alternative concurrent schemes which employ processes of coarser contexts are called *coarse-grained.*

The ETHER kernel consists of two parts: the first part includes all primitives that provide a complete set of built-in operations for developing distributed applications; the second part deals with encoding/decoding and compressing messages for an efficacious transportation across network links - this subject is fully covered in the next chapter. Moreover, the ETHER kernel is subdivided in two modules: the local kernel and the remote kernel as figure 2-1 illustrates. The local kernel resides on the processor which creates, initiates, and schedules multiple remote threads of control called RTC's. Upon creation, an RTC is a complete Lisp evaluator whose context contains all built-in Lisp definitions. The difference with an

ordinary Lisp evaluator is that I/O is performed via communication channels called *ethers*; the evaluation procedure also extends with the decoding and encoding stages; the incoming messages must be transformed into valid Lisp objects, whereas the outcoming evaluation results (messages) must be transformed into a format acceptable from the employed communication media (see section 3.6). An RTC operates in two modes: as a *passive* server or as an *active* process. In the first mode an RTC performs an endless round of passively listening on its input channel, evaluating any incoming message, and automatically transmitting the evaluation result back to its requester. The second mode utilizes a user-defined handler (transported via an ordinary message) whose evaluation hooks all incoming messages, evaluates them according to some user-written code, and transmits or not any generated result(s). This user handler may be either a simple function or a complete program consisting of an arbitrary number of associated procedural blocks; in any case this handler is considered a process in ETHERLISP whilst concurrency is achieved by assigning an instance of this process to each RTC and letting RTC's execute independently at their own speeds. Communication and synchronization among multiple RTC's is accomplished only via its common ancestor named local thread of control (LTC) or root; in particular, the LTC is responsible of multiplexing/demultiplexing messages to/from multiple destinations/sources, as well as for transporting and implicitly instantiating user handlers remotely. Mutual exclusion is also guaranteed because the LTC serves one message at a time.

2.3.1 Creating Remote Threads of Control

(**make-ether** *host* &key *(service 'packet)*) ⟶ *ether object* [Generic Function]

RTC's are explicitly created by the generic [1] primitive `make-ether()` which returns a first class object of type ETHER called *ether*. Actually an *ether* in a communication channel with an attached unique name to it used for distinguishing among multiple sources and destinations of messages. An *ether* may be of type *stream* or *packet* denoting that messages flow according to the semantics of TCP/IP or UDP/IP communications protocols. There are some general criteria for selecting *ether* types although there is nothing to prevent the use of both types in the same application. First, a stream *ether* is slower than a packet one due to the extra overhead required for a reliable two-way communications service (section 1.4.2). Second, a stream *ether* is more suitable for transmitting large volumes of data that flow continuously. Finally, the number of stream *ethers* a user application is allowed to use

[1] The term generic refers to a primitive's ability of executing both locally and remotely.

simultaneously is limited. This limit is set by the operating system and it is equal to the number of files a Unix process is allowed to have open simultaneously, since sockets are treated as files.

There are two basic solutions for mapping processes in a multiple processor configuration: (a) *static mapping* where the execution of each process is restricted to a predetermined processor, and (b) *dynamic mapping* where processes can be created on any processor at run-time; moreover, processes can migrate among processors depending on the overall load-state of a distributed system. The (b) alternative provides a higher flexibility and a better overall load balance of the system. Although RTC's can be mapped both statically and dynamically the alternative (a) is strongly recommended [2] since process migration is not supported; this implies the migration of a huge bulk of information including a process' current state, all of the received but not yet processed messages, and the network-wide propagation of the new IP address.

There are also another two alternatives to select processors for distributing processes: (a) *user controlled mapping* where the user is provided with the appropriate primitives to allocate processes on arbitrarily selected processors, and (b) *dynamic controlled mapping* where the system, based on run-time information such as the current global load of the distributed system, nominates the appropriate processors. The main advantage of the first alternative, adopted by ETHER, is that the system is less complicated.

2.4 Messages: Types and Properties

A *message* can be defined as an entity encapsulating information traveling across network links among memory-disjoint processes. In our system information is any valid Lisp object; messages may be simple or compound in structure and can be of fixed or variable and unlimited length (see section 3.7). In general, messages are in accord with the following four properties suggested by Liskov [Liskov82b].

- User programs need not deal with the underling format of messages. For example, users should not need to translate objects into bit strings suitable for transmission, or to break long messages into smaller segments. Messages of arbitrary length and complexity should be safely handled by the underlying communication system.

[2] OCCAM-2 [Inmos84] and Concurrent C [Gehani90a] realizes the static mapping.

- All messages received by user programs are intact and in good condition. For example, if messages are broken into smaller segments then the system should only deliver a message if all segments arrive at the receiving node and are properly reassembled. If the contents of a message have been scrambled the message is retransmitted.

- Messages received by a process are guaranteed to be valid objects. Support for this property requires the evaluation of the message by the sending process before its transmission. However, type checking might be required at the receiving process by the user program.

- Processes are not restricted to communicate only in terms of a predefined built-in set of types. Instead, processes can communicate in terms of values of interest to the user program.

The ETHER kernel realizes three basic message types:

- **User messages:** They are user-controlled messages carring ordinary objects among different address spaces.

- **Broadcast messages:** They are like user messages but the encapsulated objects are simultaneously transmitted to all connected receivers (RTC's).

- **System messages:** They are controlled by the underlying kernel. The transferred data include synchronization information, error messages, and messages for initializating and terminating RTC's.

2.4.1 Constructing Messages

(**neval** *expr*) —→ *expr* [Generic Special Form]

(**qeval** *expr*) —→ *(quote expr)* [Generic Function]

(**make-msg** &rest *exprs*) —→ *any* [Generic Function]

The ETHER kernel provides a flexible way for constructing messages dynamically in the sender's address space, whilst the remote ETHER kernel based on the received message contexts always generates valid Lisp expressions; the sections 2.9 and 3.5.5 cover the kernel's actions in case of erroneous transmissions. Messages are always constructed in the sender's address space; recalling that in ETHERLISP distinct address spaces may differ in context, all or some components of a message may be or may be not defined in the receiving lexical environment; likewise, some message components may be desired to be transmitted by name

rather than by value. For example, suppose that the list object `lst` and the function `foo()` whose definition is `(defun foo (lst num) ... )` are globally defined (see section 2.7). Suppose also that `foo()` is to be manifoldly transmitted to some RTC each time with a different value of num and the same value of `lst` as parameters applied during the remote evaluation process. Obviously, the *i*th value of num must be somehow enclosed in the *i*th message at run-time, whereas the (remotely defined) value of `lst` should not be included; this is vital in case of a huge list object. The above message can be constructed with minimal encoding, decoding, and transmission burdens as:

```
(make-msg 'foo (neval lst) num) ⟶ (foo lst 5)
```

The expression at the right of the arrow indicates the expression generated at sender's site and the one applied to a remote evaluator. The expression is perfectly valid since `lst` is replaced by its dynamic value attained from the remote lexical environment. The careful reader may have observed that `make-msg()` constructs compound messages as "evaluatable" Lisp expressions, i.e. lists, and that `neval(`*expr*`)` simply returns *expr* unevaluated.

```
(make-msg 'foo (qeval 'lst) num) ⟶ (foo '(a b c) 5)
```

Suppose now that `lst` is a data list bound to `'(a b c)` in the sender's lexical environment only. Its value in a message must be enclosed as the above example shows, i.e. quoted, since any received expression of the form `(foo (a b c) 5)` will signal an error denoting that the symbol `a` is illegal as a function.

2.5 Message Passing

Distributed processes interact by exchanging messages. In the simplest case, this involves two processes; one process, called the *caller* or *client*, initiates the interaction, while the other process, called the *receiver* or *server*, waits (blocks) for the interaction. At a later time processes may exchange roles. Such an interaction is called a *transaction call*. There are two main categories of transaction calls; *synchronous*, where the caller sends messages and immediately waits for the receiver to acknowledge the messages' acceptance; then, the receiver evaluates the messages and returns some result(s) to the caller. At this point, the caller can resume execution. Obviously, a synchronous transaction call involves both a strict synchronization and a bidirectional communication channel between the interacting processes. Even in case that the receiver does not return any result, i.e. a call for synchronization purposes, the caller still waits the receiver to complete the call. On the other hand, in an *asynchronous* transaction call the caller after has sent a message immediately resumes

execution; that is, no synchronization is required since the caller does not wait from the receiver neither any acknowledgement nor result. Clearly, messages are exchanged in an independent unidirectional manner.

The ETHER kernel supports the asynchronous message passing [Gehani90a] model mainly due to the achievement of maximum parallelism. More precisely the advantages of the synchronous approach can be abridged as following:

Simplicity and understandability: Synchronous message passing is easier to understand and implement than asynchronous message passing; the caller is always informed that the message has been delivered or not and a result has been received.

Bidirectional message transmission: Usually in a concurrent program processes interact bidirectionally; for instance, a client process requests a service from a server process.

Synchronization: It is very difficult to distinguish between communication and synchronization, since interacting processes can be in synchronization with synchronous send and receive primitives.

Reliability and efficient error management: Concurrent-related logical errors, due to the application program or failures owing the underlying communication media, can be reported as soon as they detected with synchronous primitives. Hence, the proper actions may be performed and the system becomes more reliable.

Straightforward debugging: One of the main drawbacks of concurrent programs is that it is very hard to debug them. With synchronous message passing erroneous situations, such as deadlock, may be easily discovered and coped with at a very early stage, either at run-time, or when looking through the program's source code.

In rebuttal, the asynchronous approach of message passing has its own advantages:

Maximization of parallelism: The chief advantage of asynchronous message passing is that it maximizes parallelism; this can be achieved in two ways: (a) real concurrent processes, executing on separate physical processors, are loosely depended since they can send and receive (and evaluate) messages in any way they want; (b) faster overall execution times due to the lack of extra overhead for error checking and (receive) overheads due to different processors execution speeds; the caller can immediately resume execution after it has sent a message. This is very important for interactions with synchronization, for instance, whenever the caller does not expect any reply.

Flexibility: Synchronous primitives can be easily simulated with asynchronous ones; the caller sends a message and immediately waits for a reply, which can be considered as an

acknowledgement that the receiver has accepted and evaluated the message. The opposite is also possible but complicates and obscures the program structure, since it involves an intermediate buffer manager process between the caller(s) and the receiver(s), requiring additional overhead for queuing and unqueuing messages and context switching.

Broadcast service and message priorities: Asynchronous message passing allows multiple-send and multiple-receive primitives, or more precisely, a caller may send to everywhere and a receiver may receive from "everywhere" ; the only restriction is that the sending processes must know only the addresses of the receivers that implies a loose connection among them. However, a broadcast facility may considerably increase the overall program's performance; for example, a disk server may accept multiple messages which can be scheduled in order to minimize the movement of the disk head. Similarly, if a server can not compute a message, because some resources are not available, it may try for another message(s) without blocking. Priorities may also be included in messages, so a server may serve requests in a user predetermined order.

Deadlock robustness: Although deadlock is hard to be detected when asynchronous message passing is in use, in some cases, it can be prevented. For instance, if process P_1 sends a synchronous message to P_2 where at the same time P_2 sends a synchronous message to P_1 then the two processes will deadlock; that is impossible if the processes communicate via asynchronous message passing.

2.6 Message Operations

(**push-ether** *expr* &optional *eth*) $\longrightarrow$ *t* [Generic Function]

(**listen-ether** &optional *eth*) $\longrightarrow$ *any* [Generic Function]

(**ether-readable-p** &optional *eth*) $\longrightarrow$ *nil* | *t* [Generic Function]

Liskov [Liskov79a] has proposed three possibilities for sending a message: (a) a no-wait send, (b) a synchronized send, and (c) a remote invocation send. The primitive **push-ether()** provides the first alternative asynchronous send operation based on the buffering mechanism supplied by both TCP/IP and UDP/IP protocols. However, only TCP/IP ensures that a message has been delivered by blocking a speedy sender when there in no room in the receiving queue; since UDP/IP does not provide such a facility UDP/IP messages may be lost unless some user code prevents queue overflow. The difference between a synchronized send and a remote invocation send is that in the former case the sender waits until the message has been accepted, whereas in the last case the sender waits until it receives a

reply. ETHER supports (a) and partially the (b) sending approach based on TCP/IP whilst the (c) alternative can be easily implemented as a send immediately followed by a receive operation. ETHER permits multiple TCP/IP (stream service) or UDP/IP (packet service) active channels to be utilized by a single application; the generic sending primitive automatically recognizes and attaches the proper communications protocol for every transmission. Andrews mentions that "...Channels are like semaphores that carry data. Hence, the send and receive primitives are like the V and P operations, respectively." [Andrews91, p:343]. According to this remark synchronization between memory-disjoint interacting processes is achieved straightforwardly by a blocking receive operation. In ETHERLISP this operation is accomplished by the `listen-ether()` primitive. However, an optional non-blocking receive service can be provided in terms of a *selection* over a collection of channels by returning those channels having pending messages and fulfill some criteria. Our system supports selection via the `ether-readable-p()` predicate primitive whose usage is illustrated in section 2.11.3.
Other message operations include broadcast, timed, and peeked receives. A broadcast message is received by every active RTC including LTC but any potentially generated result message is swallowed remotely; that is, an RTC after the evaluation acknowledges the sender and immediately switches to the next receive operation without returning any result, except when the message causes an error. The reason of this design is because broadcast messages are mainly used for initializing, initiating, or synchronizing processes. Timed and peeked receives are usually employed for scheduling purposes; this will be covered in chapter 6 where the scheduling policy of our concurrent model PRAXIS is discussed in details; for now, a timed receive blocks the receiver until either a reply arrives or x time units have been elapsed, while a peek operation returns but not consumes a message so that a subsequent (ordinary) receive operation will see the same message.
All fundamental message operations mentioned above but broadcast can be applied from a remote site too. This possibility provides remote user-developed handlers with a flexible and independent way of coping with messages.

2.7 Initializing a Distributed Application Program

(init-ether *file* &rest *eths*) $\longrightarrow$ *t* [Function]

Any parallel algorithm in ETHERLISP presumes the existence of an appropriate number of remote threads of control. The algorithm is loaded in the LTC's address space and concurrent execution starts immediately after code and data have been dispersed (and probably

partitioned first) among RTC's. After an RTC has been created its address space is considered *empty* by means that there is no indication of any algorithm-dependent definitions. For instance, any message of the form `(userfun arg1 ... argN)` being evaluated by a newly created passive RTC (section 2.3) signals an error because `userfun()` is undefined in the receiving address space. The problem is overcame due to the ability of sharing files via *NFS* [SunOS] over one or more networks. The primitive `init-ether()` causes a file containing an application being executed to be loaded and evaluated by all RTC's simultaneously. Initialization also serves for other purposes discussed in the next chapter.

2.8 Object Properties in a Distributed Address Space

The transportation of objects from one address space to another imposes the creation of copies since pointers to these objects are meaningless beyond the scope of the sender's address space. Hence, mutable objects (in COMMON LISP all objects but numbers, characters, and symbols are mutable) are not `eq` to their remote copies.

In figure 2-2, a fixnum and a simple vector are bound to `int` and `svec` respectively at the address space of the root (LTC) process. The first comparison returns `t` because the number 123 is a non-mutable object. The second comparison returns `nil` because the vector bound to `svec` and its remotely evaluated copy are two different objects; whereas, the third comparison returns `t` since `svec` and its copy are objects of the same type. In the last case, both binding and comparison is performed remotely; the returned value is `nil` because `svec` and its copy, sent from root, are the same but not identical objects remotely as well.

2.9 Global Error Handling

There are several failures which may occur when processes communicate across network, including the possibility of partial failure (host crash), a process is unable to receive messages, messages have been garbled, or the syntax of the (LISP) system has been violated at a distant location. In all cases the user is informed by an appropriate error message. In particular, any remote error message is prefixed by the symbol `Remote`, the name of the remote processor, and the unique identification number of the channel. The LTC signals the local error handler whenever it encounters an error header in a received message whilst the rest of the message specifies the cause of the error.

```
                                                                    >(setq :
⟶ #(A B C)
>(and (push-ether int eth) (eq int (listen-ether eth))) ⟶ T
>(and (push-ether svec eth) (eq svec (listen-ether eth))) ⟶ NIL
>(and (push-ether svec eth) (equalp svec (listen-ether eth))) ⟶ T
>(and (reval (make-msg (neval setq) (neval svec) svec) eth)
      (push-ether (make-msg (neval eq) (neval svec) svec) eth)
      (listen-ether eth))⟶NIL
```

Figure 2-2: Mutable and non-mutable object handling.

2.10 Process Termination

It is of common logic that a concurrent program terminates when all of its co-operating processes has finished executing. However, a number of problems arise. Consider a parallel searching algorithm where a number of processes search for a specific key, each within a different portion of a sequence, and another process collecting the results. Assuming that the key is not unique there are two solutions of the problem: (a) the collective process terminates as soon as it receives a result. This solution maximizes parallelism but the remaining processes become deadlocked since they are blocked for ever trying to return a, positive or negative, result to the collective process that has been already terminated. (b) the collective process reports the result as soon as possible but waits until all processes have returned a result. The second solution clearly entails to a correct and safe concurrent programming.

A strict notion of process termination does not lay in ETHERLISP; in the most complicated case a process in this system is an application-dependent function hooking successive messages directed to it. A process is considered completed immediately after the sending of its last result message; the RTC hosting such a hook function is still active and ready to serve simple messages or additional hook functions. This procedure continues until an RTC evaluates a *kill* system message in which case the channel is closed and removed from the kernel's active channel queue `*ether-list*` (see section A.3).

2.11 Examples

The main goal of this section is to illustrate the manner various networking operations can be implemented when the basic built-in primitives are combined properly. More examples are given through out this thesis but in any case the reader should be in close contact with

appendix A which contains a complete user guide.

2.11.1 On Establishing a Remote Login Session

The *rlogin* command available in the networking software installation of UNIX can be straightforwardly coded in ETHERLISP as figure 2-3 illustrates. The function **rlogin()** starts a remote login session (a passive RTC) from the local root process (LTC) to the remote machine named *coign*. The procedure is totally transparent, is that the LTC sends any expression read locally to be evaluated remotely, and the result (pointed by an arrow) is automatically displayed on the terminal. The remote session might look like the following series of instructions:

```
>(rlogin "coign")
coign>(defun sqr (x) (* x x)) ⟶ SQR
coign>(sqr 4) ⟶ 16
coign>(exit) ⟶ Connection closed.
>(sqr 2) ⟶ Error:  The function SQR is undefined.
```

Any attempt to evaluate **sqr()** after the remote session has been terminated results to an error, since the function was defined only in the remote address space. Note that the *ether −rhost−* is not maintained in the internal channel queue ***ether-list*** since it is created in the non-permanent lexical scope of a **let()** construct.

```
(defun rlogin (host-name)
  (when (not (verify-host host-name)) (error "The host ~S is unknown." host-name))
  (let ((-rhost- (make-ether host-name :service 'stream))
        (fin-expr '(exit)))
    (loop (fresh-line *trace-output*)
          (format *trace-output* "~S>" (cdr (ether-hosts -rhost-)))
          (let ((expr (read)))
            (when (equal expr fin-expr) (kill-ether -rhost-)
                  (return-from rlogin (format t "~%Connection closed.")))
            (push-ether expr -rhost-)
            (fresh-line *trace-output*)
            (format *trace-output* "~S" (listen-ether -rhost-))))))
```

Figure 2-3: Code for the *rlogin* Unix command.

2.11.2 Selective Remote Bindings

(**setr** *var value* &rest *eths*) ⟶ *any remote binding* [Special Form]

In many occasions, a more indicative, compact, abstract, and elegant way of binding objects remotely is required (see section 4.4.3). The built-in special form **setr()** assigns the value of

value to the dynamic variable named by *var* in the lexical environment of the RTC's denoted by *eths*; when *eths* is omited the binding is broadcast to all active RTC's. The returned value is the value assigned that also acknowledges a successful remote binding. Figure 2-4 demonstrates a selective remote binding facility similar to the built-in `setr()`. Selection is determined by the parameter `elist` specifying the address spaces in which bindings are to be performed. Precisely, when `elist` is null the elements (channels) of `*ether-list*` are the implicit binding destinations in which case a broadcast is performed. When `elist` is a subset of `*ether-list*` separate type checking is required to prevent differentiations in the contents of remote lexical environments in case `elist` contains an element that is not of type *ether* and some bindings have been already completed.

```
(defun setr2 (&key (fun 'setq) ; Default binding operator
                   var         ; Variable's name
                   value       ; Value being assigned
                   elist)      ; Broadcast to all or some active RTC's
  (when (not (typep var 'SYMBOL)) (error "~S is not a symbol." var))
  (when (endp *ether-list*) (error "There are not active ethers."))
  (let ((msg (cons fun (cons var (cons value nil)))))
    (if (endp elist)
        (ether-broadcast msg)
        (progn (dolist (-e- elist)
                 (when (not (typep -e- 'ETHER)) (error "~S is not of type ETHER." -e-)))
               (dolist (-e- elist) (push-ether msg -e-) (listen-ether -e-))))) value)
```

Figure 2-4: Selective remote bindings.

2.11.3 Scheduling and Synchronizing Distributed Processes

(select-ether &key *(block t)* **)** $\longrightarrow$ *nil* | *ether object* [Function]

Execution of multiple concurrent processes gives rise to the need for scheduling and synchronization. In concurrent programs processes must interact in mutual exclusion ensuring that critical sections are not (erroneously) accessed simultaneously. Moreover, proper scheduling entails the optimal utilization of the system's resources. The built-in construct for the accomplishment of these purposes in ETHERLISP is the `select-ether()` operation that picks up and returns a readable (with pending messages) *ether*. Selection blocks when there is not any readable channel unless the keyword *:block* is set to `nil`. Selection can be requested according to several channel properties such as message traffic (ether load), or scheduling schemes such as the Round-Robin policy.

In figure 2-5 selection is based on the message traffic of multiple channels. In general, the

```
(defun select-ether (&key max-load min-load (block t))
  (let ((ready-ether) (imax-load 0) (imin-load most-positive-fixnum))
    (loop (if ready-ether (return ready-ether))
          (dolist (e *ether-list*)
            (when (ether-readable-p e)
                  (let ((cur-load (ether-load e)))
                    (cond ((and max-load (> cur-load imax-load))
                           (setq imax-load cur-load) (setq ready-ether e))
                          ((and min-load (< cur-load imin-load))
                           (setq imin-load cur-load) (setq ready-ether e))
                          (t (setq ready-ether e))))))
          (if (and (not ready-ether) (not block)) (return nil)))))
```

Figure 2-5: Load-based criteria for selecting readable communication channels.

load of an RTC can be roughly estimated from the number of received, processed, and returned messages. Assuming that efficiency is related on an even distribution of computation a scheduling scheme can be based either on channels with light traffic (overloaded RTC's) or on heavy traffic; in the last case a processing equilibrium among RTC's can be achieved by assigning additional work-load to lightly loaded RTC's.

2.11.4 Synchronization via Monitors

Andrews and Schneider [GAFS83] define monitors as a collection of permanent variables used to store the resource's state and some procedures which implement operations on the resource. Erroneous access to shared resources is prevented since the execution of the procedures in a monitor is guaranteed to be mutual exclusive. Monitor constructs can be easily implemented in ETHERLISP. In figure 2-6 a typical example is presented where two processes share a buffer; one process, the `producer()`, fills it whereas the second one, the `consumer()`, evacuates it. `monolithic-monitor()` encapsulates the buffer (shared resource) and the operations (functions) performed on it. Mutual exclusion is guaranteed since the monitor accepts and serves one message at a time. Any message containing an integer triggers its insertion into the buffer; whereas any message of the form **(get)** entails the activation of the corresponding function. Note that both operations are suspended, say for one second, if the buffer is either full upon an insertion request, or empty upon a get request.

```
(defun producer () (loop (let ((datum (random 100))) (push-ether datum))))

(defun consumer () (loop (push-ether (make-msg (neval get))) (listen-ether)))

(defun monolithic-monitor ()
  (let* ((buffer-size 100) (n 0) (in-ptr 0) (out-ptr 0)
         (buf (make-array buffer-size)))
    (labels ((insert (datum)
               (when (= n (1+ buffer-size)) (sleep 1))
               (setq (aref buf in-ptr) datum)
               (if (= (incf in-ptr) (+ buffer-size 1)) (setq in-ptr 0)) (incf n)))
    (labels ((get ()
              (when (= buffer-size 0) (sleep 1))
              (let ((datum (aref buf out-ptr)))
                (if (= (incf out-ptr) (+ buffer-size 1)) (decf n)))))
    (let* ((eth (select-ether)) (request (listen-ether eth))
      (if (integerp request) (insert request)              ; Insert datum.
                             (push-ether (get) eth))))))) ; Get and send datum.
```

Figure 2-6: Construction of a monolithic monitor.

2.12 Summary

In this chapter we presented ETHERLISP's kernel that provides COMMON LISP with the basic primitives for developing intergrated concurrent applications. However, fundamental aspects such as process migration and persistent communications via recovery mechanisms are not (currently) covered; our present goal is the extraction of the highest possible parallelism which can be achieved only from a system that produces the minimal overheads. Assuming that the reader has been consulted appendix A he/she observes that the ETHER kernel consists of a plethora of primitives. We believe that many primitives that perform the absolutely necessary tasks instead of generic ones contributes to a considerable reduction of the overall system's execution overhead.

Chapter 3

On Minimizing the Network Overhead

3.1 Introduction

IN THE PAST FEW YEARS numerous distributed systems have been developed, while other such systems are still being developed today. One common feature of most of these systems is the message passing concept (section 2.5). Processes execute concurrently on distinct processors connected via a network, and the only way of communication is by exchanging messages. One of the main drawbacks of physically distributed systems is the large communication overhead, since the transmission times of the communication protocols are still measured in milliseconds. Furthermore, additional overhead is required for encoding/decoding messages since they carry data structures that fulfill the semantics of a distributed system, semantics that must be preserved until receipt at a distant location. Consequently, a considerably large amount of computational power is consumed, passively in some sense, for communication purposes. In this chapter we present the second part of the ETHER kernel FILOS, standing for ForwardIng Lisp ObjectS, a mechanism which filters interchanged messages for reducing their size and the communication cost in general.

3.2 Objectives

The primary objectives of this chapter are stated in the next conjecture:

■ **Conjecture 2** *The network overhead of a distributed system involves some kind of information of the interest of a user interchanged among physically separated processes, the preservation of the semantics of the transferred data, as well as any overhead caused by both software and hardware of the underlying data delivery mechanism (see chapter 1).*
A drastical and efficient compression of interchanged messages' length results to a significant improvement to the overall performance of a distributed system, whereas critical factors such as the delay, busy and idle periods, utilization, capacity and reliability of a circuit are affected positively as well.

3.3 Analysis of the Network Overhead

The asynchronous message passing model has been adopted for interprocess communication among *ethers* (see section 2.5) for many reasons However, the main disadvantage of this model is that it requires an intermediate buffering of the transmitted messages. Since buffering is not supported by hardware, the source programming language via operating kernel's interrupts should manage it either by buffering at the message's destination process, or by an intermediate dedicated buffer manager process. In our network cost analysis, we estimate the overhead required by a bidirectional asynchronous interaction of two endpoints of a channel executing on distinct processors. This implies two *send* and two *receive* operations; the cost required for an acknowledgement message has been ignored, since it may be assumed that the receiver's buffer is always available. The factors that produce the highest burden include:

C_{Ctx} **Context switch**: Assuming that there are more than one user processes per processor, the underlying scheduler determines the next process to execute; this implies the saving of the state of the current executing process and the restoration of the chosen one. We assume that the process of our interest is always the next scheduled process.

C_{Cpy} **Message copying**: A message being transmitted must be first copied from the sender's context space to the message manager, or from there to the receiver's context.

C_{Buf} **Message buffering**: A dedicated queue manager is responsible for allocating buffer space for messages, accepted but not received, at the receiver's side, or at the sender's side messages that the receiver is unable to accept due to lack of queue space. After the message has been received the buffer must be freed.

C_{Net} **Network overhead**: The transportation of a message through a physical network link requires a number of operations, which mainly include the writing and reading to/from input/output ports, the segmentation of large messages into smaller ones, the encoding/decoding of messages into the appropriate formats, and the acquirement of the permission to send/receive messages to/from the network manager.

C_{Trn} **Data transmission**: This overhead denotes the actual data transmission through physical network links. In this case the cost is given by the equation:

$$T_C = C_N * (C_{BN}/T_R) \tag{3.1}$$

where, C_N is the number of the transferred characters, C_{BN} is the number of bits per character, and T_R is the transfer rate of the communication media measured in bits per second (bps). Thus, the last fraction indicates the transmission time of one character. In theory, the transmission time of a 10Mbits/sec Ethernet is (roughly) one millisecond per byte.

According to the above factors, the (rough) cost of sending n messages between the local thread of control (LTC) and a remote thread of control (RTC) is:

$$C_{TC} = \sum_{i=1}^{n} 4C_{Ctx} + 2C_{Cpy} + 2C_{Buf} + 2C_{Net} + 2C_{Trn} \tag{3.2}$$

The context switching C_{Ctx} is expensive on workstations with networking facilities because there are additional processes (deamons), such as the electronic mail and the network file system (NFS) [SunOS]. Moreover, the scheduler needs information for processes blocked on a receive operation; this imposes an additional context switch for the message manager, and hence four C_{Ctx}'s are needed for each message. The rest of the factors charge the system twice, once at each endpoint of the communication channel. However, the network overhead C_{Net} causes the longest delay since it is measured in milliseconds; for instance, on the VAX 8650 LAN configuration C_{Net} is about 8 milliseconds, C_{Ctx} is 53 microseconds, and a function call is 4.2 microseconds [Gehani89b]. In addition, Leichter [Leich89, p:53] measured 1.87 and 5.78 seconds as the mean elapsed round-trip time for 500 packets of size 50 and 1400 bytes respectively sent from a VAX 8600 to a MicroVAXII and echoed back. Although C_{Ctx} and C_{Net} are considered constant costs, the other costs can be reduced when

C_{Trn} is reduced by an encoding/decoding mechanism which efficiently compresses messages. More precisely, when the quantity of data decreases then: (a) T_C obviously decreases; (b) C_{Cpy} decreases since less data are copied; (c) C_{Buf} decreases since the message manager manipulates smaller messages; meanwhile, the limited [1] buffer space [2] "increases" by means of queuing more (compressed) asynchronous messages; moreover, the system becomes more reliable since the probability of a message to be lost, due to lack of buffer space in case of a UDP/IP channel, or the acknowledgement delay in case of a TCP/IP channel, decreases.

3.4 Issues on Compression of LISP Object Structures

The main observation of the communication cost analysis indicates that the network overhead is influenced by several factors strongly depending on the underlying network system such as, the latent transmission time, and the unpredictable load of the participating processors. On the other hand, there are factors depending on the actual messages, such as their size, structure, and semantics. These factors have been classified into two categories; those factors that influence messages physically, and those that influence them logically. The first category includes the *physical compression* concept that can be viewed as a process of reducing the data quantity contained in messages before their transmission, and the expansion of such messages into its original format upon receipt at a distant location. The second category is strongly related on the semantics of LISP, and refers to the *logical compression* concept that can be viewed as a process of reducing the number of times that Lisp objects need to be created, looked up, or interned upon receipt at a distant location. When a non-mutable object, such as a number, a character or a string, is transferred by value (transmission by name is also possible as we shall see), a fresh structure of the appropriate type to hold that value must be created in the receiver's address space. For instance, upon receipt of a literal simple string object the receiver must create a structure which holds information such as the object's type, dimension, length, and contents; besides, compound strings require additional information since they may be displaced to another string,

[1] The buffer capacity of our system is limited to $64K$ bytes, although the actual amount is less; for each message additional information, such as the sender's IP address, message priority, and a pointer to the next message (*mbuf chain*), are stored as well. Therefore, a very large buffer capacity could slow down the execution time of a process due to larger paging and swap times.

[2] According to the Sun implementation of communication protocols [SunOS, vol:10] the buffer space is handled by the memory manager which employs three kinds of data storage (mbufs): (a) the *small* mbufs are the fundamental type of constant capacity 112 bytes and they are guaranteed to start on a 128-byte boundary; (b) the *cluster* mbufs are of fixed size $1K$ and provide storage for larger amounts of data whilst one small mbuf is required to refer to a given cluster; (c) finally, the *loaned* mbufs are handled by the system's separate allocator which is also responsible for freeing them.

may have a fill pointer, or may be adjusted their size. The receiver can straightforwardly create permanently in its context such a structure when the object's type and contents are provided in a single message or in a logically separated block in a compound message.

In rebuttal, the structure of a symbol is more complex; it embodies information for the type of the object, the print name, the length of the print name, the property list, the value of a dynamic binding, a special form definition, a global function definition, the home package, and some other less important information. A couple of problems arise when such a structure is to be literally transmitted, since the encapsulation of some of these information within messages is or might be very expensive, redundant, and sometimes impossible. For example, in case of the built-in symbol `car` the encapsulation of the property list, holding the function's documentation text accessible by the `help()` primitive, is redundant because it is not needed by the receiver's evaluator, whilst the global function definition, holding the executable code of the primitive's body, is compiled code and hence, impossible to be encapsulated into a message. Instead, suitable included information should guide the receiver to locate in its address space the base memory address pointing to the compiled code of the primitive's body. Another alternative is the transmission of the symbol's print name; this requires the searching of all the internal and external (remote) system packages, and the replacement of the "raw" print name with its definition. On the contrary, the executable code of a user-defined (interpreted) function can be easily enclosed in a message, since it is an ordinary compound list object; obviously, this solution is very expensive especially when functions with large bodies are transported frequently (see section 5.5).

Apart from symbols whose structures are defined in the sender's context there are symbols, for example the local variables in the lexical scope of a `let()` construct, which are defined and used only during the evaluation process. The later implies that the only useful information known prior to transmission is the symbols' print names. For instance, consider the expression `(setq x (aref #(a b c) 0))` which is a legal Lisp expression but all of the user-defined symbols prior to evaluation are unbound, or more precisely, bound to the special object `#<OBJNULL>`; any attempt to evaluate the vector's elements as individual objects signals an error. Although the above expression may successfully transmitted and evaluated in a different address space, the encapsulation of any other information but the symbol print names, generated by the sender's Lisp reader, is redundant.

3.5 Logical Message Compression

The logical message compression constitutes the main concept of FILOS; it contributes to the highest reduction of the transferred data quantity, and the minimum time for the settlement of compressed (encoded) received objects. The chief concept is based on several tables where indices "pointing" to Lisp objects being transmitted are maintained on. This hints that objects are transmitted either explicitly, whenever they encountered for the first time, or implicitly by transmitting their associated indices. Upon receipt of a message, the decoding process determines whether or not an object needs to be created, looked up, or interned by examining if it is a new object or an index (previously received and hence known object) respectively.

3.5.1 The System-Index Table

The first table is called the *system-index* (SYS_i) table and its need is based on two particular features of COMMON LISP. First, LISP is a very rich language in the sense that it consists of a fairly large number of built-in objects, including functions, macros, special forms, special variables and constants. Second, these built-in objects (in interpreted mode) are executable Lisp code as well. Since all these objects are accessed to by symbols, are prior known (defined) on all RTC's upon creation, and their occurrence in a message as symbolic names is extremely high (section 3.9.3), we can transmit just constant indices, representing these symbols, instead of their structures or even their print names; in this way, any built-in object is not created, looked up or interned in a remote lexical environment. The cost of the table in main memory space is insignificant, since the total number of all built-in primitives counted in our AKCL is 863 and each table's pointer is a 32-bit integer; thus, the memory consumed is 3452 bytes. For reasons of efficiency, we altered the internal structure of a symbol by adding two extra fields. The first field, the index field, holds the (permanently associated) index of a built-in symbol, whilst the second one, the type field, indicates the type of the table a built-in symbol is maintained on. The types of the added fields in ANSI C notation are a *short integer* (2 bytes) and a *char* (1 byte) respectively; hence, the additional requirements of main memory are 2589 bytes for all built-in primitives.

The SYS_i table is built once during the initial compilation, at C level, of the language where each built-in symbol's index field is assigned a number of increasing order starting from zero, whilst the type field is assigned a small integer denoting that the symbol is a built-in one. Then a pointer is saved on the table at a location equal to the index field of the

symbol. Consequently, any occurrence of a SYS_i index in a message upon its receipt results to the automatic and instantaneous replacement by the related primitive. Furthermore, the cost of searching the table is nil since each entry is randomly accessible by the index field. However, some built-in primitives such as `make-array()` and `sort()`, must be inserted onto the SYS_i table manually prior to the compilation process, since they have been implemented on top of LISP and hence unknown to the CC compiler [BKDR88, SHGS87].

3.5.2 The Hash-Index Table

Another category of objects defined prior to any transmission are contained in the code of a (loaded) Lisp application being executed. These objects include any user-defined function, macro, special variable and constant. All these objects are handled by another table, called the *hash-index* (HSH_i) table. HSH_i is an actual hash table implemented in C and whose each node consists of two fields; an index field (16-bit) for locating the table's entries, and another index field (32-bit) pointing to any user defined object listed above. Obviously, the requirement of main memory is unknown since the table's size is unpredictable depending on the size of the application.

The HSH_i table is built at run-time in the address space of all RTC's upon their initialization (section 2.7). After the initialization process has been completed, any user-defined symbol is properly updated; the type field denotes that the object is handled by the HSH_i table, whilst the index field is assigned a number of increasing order starting from zero. Although this table requires searching, the overhead produced is insignificant when the size of the table is small as is usually the case. Experiments show that even for large scale applications it is not important; this is proved by a Lisp program analysis presented in section 3.9.3.

3.5.3 The Atom-Index Table

A last category of possibly transmitted objects includes all these objects that are bound to non-mutable values, or values created in the lexical environment of different RTC's during the lifetime of an application. These objects are manipulated by the *atom-index* (ATM_i) table, which is structurally identical to the SYS_i table but its size is restricted (see sections 3.7.2 and 3.9.4) to 256 locations resulting to an occupation of $1K$ of main memory.

The table is updated at run-time by a pair of memory-disjoint processes (RTC's) each time they exchange an asynchronous message. Both RTC's update their private ATM_i tables

"simultaneously" by means that the same exactly actions must be performed for retaining the involved ATM_i's in the same state. Note that although the ATM_i contents of two interacting processes must be the same, their lexical environments may differ. For instance, in case of a message including a mutable string twice the sender's encoding process creates an ATM_i pointer to this string structure, and the string's characters are literally written in the network buffer; consequently, the second occurrence of the string is encoded as an ATM_i index instead. Upon receipt of the message the receiver's decoding process creates a string structure whilst an ATM_i pointer to this structure is created; this pointer must be referenced via an index of the same numerical value at both sites, so the received encoded ATM_i index must be replaced by the already created remote string structure. Similar actions are performed for simple identifiers, such as the variables in the lexical scope of a `let()` construct, which are symbols bound to the special object `#<OBJNULL>` before the evaluation; in particular, the index and type fields of such symbols are properly updated, and hence, the ATM_i table is accessed by these values without any searching overhead. Obviously, the limited ATM_i size implies the overflow of the table at some future time. In that case, the table is *cleared* simply by start recounting from zero. After ATM_i has been cleared all of its contents are considered fossils, and therefore, any objects previously encoded as ATM_i indices are literally re-encoded. The later indicates that the state of the ATM_i table of both sender and receiver remain the same no matter how many times the sender has cleared its ATM_i table. ATM_i content mismatch is also prevented even when the sender has sent numerous messages whilst the receiver has not received any of them. The fact that the contents of the ATM_i table are retained, while the table is not cleared, presumes the existence of a history mechanism. Strictly speaking any objects included in messages being transmitted at different times are encoded as ATM_i indices if there are ATM_i pointers ponting to facsimile object structures (see section 3.9.4).

3.5.4 A Graphical Presentation of the Index Tables.

The concept of all index tables is pictured in figure 3-1 where we assume that several messages have been exchanged prior to the transmission of the message which is under examination. This is denoted by the first and second ATM_i indices that point to a symbol and a string defined somewhere in the address spaces of both processes. These indices is the history attained up to this point; for instance, any future message that contains a symbol identical to the structure of a results to the encoding of the ATM_i index 0, whilst

its creation an internment at the receiver's side is not performed. The built-in primitives `defun()` and `*` are handled by two permanent SYS_i indices. Note that the actual entity pointed to by the index 536 is the built-in special form defined in the address space of each evaluator (RTC) upon creation; hence, the receiver's decoding process simply replaces the index 536 with the associated (remote) definition. In case of the user-defined function `bar()` a permanent HSH_i index points to the related definition (in interpreted mode). This definition is also known in the receiver's address space during the initialization phase mentioned in section 2.7. The symbols `foo` and `int` (first occurrence) are encoded as literal print names since they are encountered for the first time; these objects which are called *new-objects*, the parentheses, and the print names are encoded as it will be mentioned in section 3.7.1.

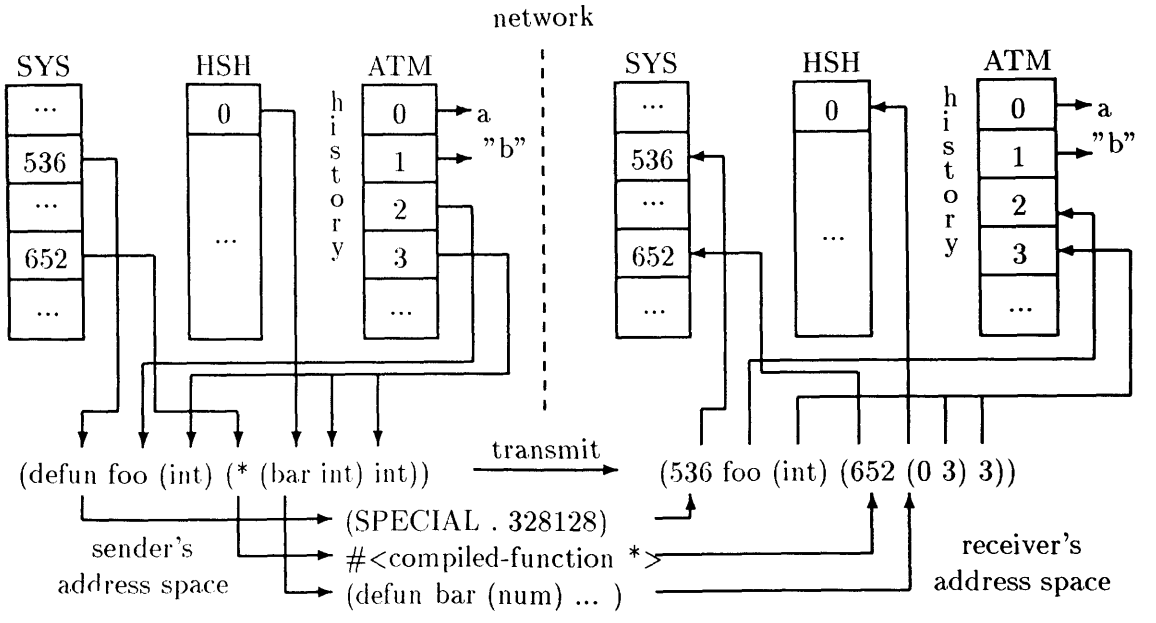


Figure 3-1: A Graphical Presentation of the Index Tables.

3.5.5 Miscellaneous Issues on Index Tables

The fact that the size and contents of the SYS_i table remain constant results to its inexpensive maintenance and access; in addition, the table is automatically inherited from the root process to any RTC upon creation. Conversely, the main drawback of the HSH_i table is its unpredictable size and the larger stored information, along with a rather expensive hashing

performed for each reference of a hash index. The ATM_i table is of great importance since it usually carries out most of the compression. However, there are some issues that obscure its function and hence require further explanation and confrontation.

- **Unique State**: It has been mentioned that the contents of the ATM_i tables of two interacting endpoints of a communication channel must be the same after the completion of the transportation of a message. This can be achieved only if each endpoint maintains its own ATM_i table along with a pointer, called *state* pointer, specifying the current table's position. Moreover, another pointer is required pointing to the position that the encoding or decoding operation of the last message was successfully completed. Thus, any error occurred during the sending or encoding stage of a message causes the interruption of the procedure, and the movement of the state pointer to the previous (safe) position. In case that an error occurred remotely during the evaluation of the message, the sender is reported with a system error message and the ATM_i table state of both endpoints is similarly restored.

- **Garbage Collection**: The contents of the ATM_i table are vague upon creation of an RTC. Hence, the table is initialized and each entry points to the special Lisp object `#<OBJNULL>`. When at a later time the table runs out of space, it is *cleared* by simply moving the state pointers to the first location of the table. Since the old contents are no further meaningful (fossils) the table is initialized again.

- **Frugality**: The overhead due to searching this table is insignificant since symbols are randomly accessible by their index field values. This results to a significant data reduction especially for symbols with long print names. However, objects like small fixnums and characters are always literally encoded, since the indexing benefits may be negligible compared to the searching overhead.

3.6 Physical Message Compression

Anything being transmitted via Ethernet must be a string of characters of limited length. The transmission of a message imposes the sender to *encode* Lisp objects into characters and the receiver to *decode* them, e.g. to transform raw characters into valid Lisp data structures. The physical message compression is the second concept of FILOS attempting to reduce the data quantity whenever that is possible; this includes the elimination of special characters which identify objects, such as the surrounding quotes of a string and the parentheses of a

list, the hash and the surrounding brackets of a vector, the blanks between the elements of lists or vectors used for the print representation of objects by the Lisp *writer*, and finally, the conversion of numbers in a binary format.

3.7 Variable Length Object and Index Blocks

The underlying FILOS communication buffer is a string of a limited size $3K$ bytes and it consists of variable length blocks each encapsulating a Lisp object or an index. However, sometimes it is impossible for an object, like a very long list or a vector, to be fitted in the buffer; in that case, the message is broken into segments of suitable size; the last byte of each segment holds an 8-bit *end-of-segment* block, and therefore messages of arbitrary length can be transmitted. The last flexibility does not come for free because the sender must wait for the receiver to acknowledge every segment, otherwise this may lead to the overflow of the message queue of a slow receiver. Clearly, for message lengths larger than $3K$ bytes processes in ETHERLISP interact synchronously. We believe that under normal communication requirements a $3K$ segment capacity is adequate for encompassing unbreakable large data structures, or it yields the minimum number of segmentations. Note also that the acknowledgement is sent before the receiver starts decoding and hence, the sender and receiver encode and decode segments in parallel.

Each variable length block is preceded by a 1-byte header as figure 3-2 illustrates. The last three significant bits indicate the type of the block, whilst the five most significant bits specify other information related with the encoded entity. There are eight block types:

- **Escape block**: The rest of the header contains either a left parenthesis indicating the start of a new *cons*, or a quote indicating the start of a quoted object, or a system message indicator. System messages (section 2.4) include messages for initializing or killing RTC's, sending broadcast messages, or restoring the ATM_i states.

- **Small SYS_i, HSH_i, or ATM_i index block**: The rest of the header contains the absolute binary value of a small index.

- **Large SYS_i, HSH_i, or ATM_i index block**: The rest of the header specifies the number of bytes required to represent the absolute binary value of a large index.

- **New-object block**: The rest of the header indicates the type of the Lisp transmittable object encoded in the block; consequently, up to 18 different object types can

be supported although 12 are the basic and of our interest Lisp types. Structures like *readtables* and *hastables* will be examined in the future.

3.7.1 New-object Blocks

Currently FILOS allows the transmission of most of the fundamental readable object types; an object type is readable if it can be generated by the Lisp *reader* using the type's default print representation. Any attempt of sending any non-readable, and hence non-transmittable, object signals an error and the transmission is interrupted; meanwhile, the state of the sender's ATM_i table, if changed, is rolled back to the state prior to the erroneous transmission. Next, we present the variable length structure of some exemplary objects, generated by the encoding process, in order to point out the cases the physical compression is feasible.

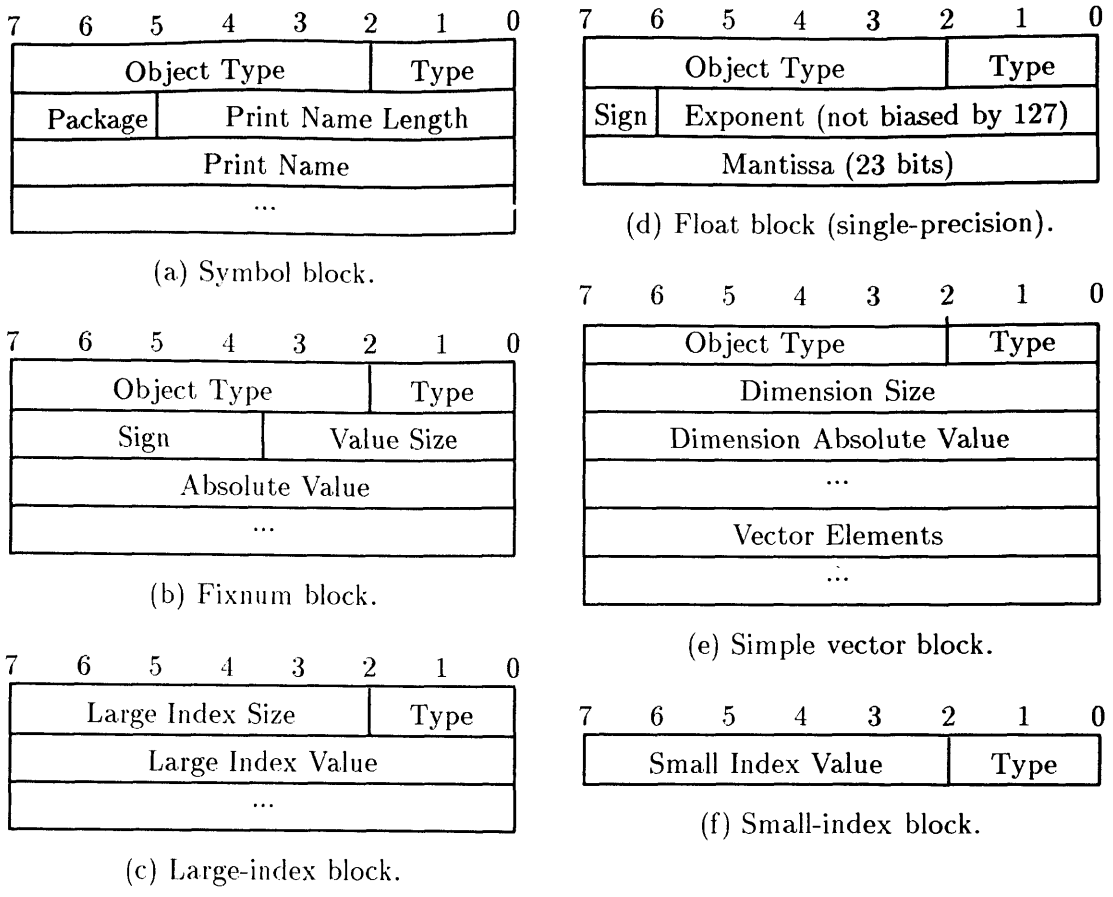
Symbol block structure: Symbols are the most frequently used, and consequently transmitted, objects whose structure is stated in figure 3-2,(a). The length of a print name is encoded within 6 bits since we consider that symbols with print names longer than 64 characters are extremely rare. The encoding of the symbol's home package eliminates the interment burden upon receipt. It should be made clear that all symbols which are dynamicly bound in the `#<"USER">` package are always encoded as objects of their dynamic binding type, except if `neval()` (section 2.4.1) is in effect. Clearly, FILOS claims two additional bytes for each encoded print name.

Fixnum block structure: The block structure of a fixnum is stated in figure 3-2,(b). The largest transmittable fixnum could be equal to $2^{32} - 1$ which is greater than the Lisp constant `most-positive-fixnum` which equals to $2^{31} - 1$. Physical compression is not feasible for fixnums smaller than or equal to five decimals; thus, fixnums greater than or equal to 10^4 are only maintained by the ATM_i table.

Float block structure: Floating-point numbers, figure 3-2,(d), are handled according to the ANSI/IEEE Std 754-1985 floating point standard adopted by the Sun's External Data Representation (XDR) standard [IRIS-4D87]. A floating-point number is decomposed into its sign S, its exponent E, and its mantissa F. Recalling that in IEEE the exponent is biased by 127 and that the radix R of a float is 2, a single-precision float is composed by the formula $(-1)^S * R^{\{E-Bias\}} * 1.F$. Physical compression is feasible for floats with decimal digits greater than or equal to two but obviously the encoding of floats is expensive.

List block structure: Lists are of particular interest because they are frequently used,

form executable Lisp code, and produce the highest physical and logical compression. The elements are enclosed between the special 8-bit blocks *cons-head-block* and *cons-tail-block* denoting the surrounding parentheses. The dot sign of a dotted *cons* is represented by a *cons-dot-block*. The blanks between elements are omited as in the vector case (figure 3-2,(e)).



(a) Symbol block.

(b) Fixnum block.

(c) Large-index block.

(d) Float block (single-precision).

(e) Simple vector block.

(f) Small-index block.

Figure 3-2: The generic structure of variable size object blocks.

3.7.2 Index Blocks

Indices are encoded into index blocks of variable or fixed length. Their absolute values are also converted into a binary format. There are two cases of index blocks:

Small-index block structure: The existence of small index blocks, figure 3-2,(f), is due to the fact that all of the index tables produce indices in the range 0 to 31 inclusive. Thus, the five least significant bits of a single byte can hold the (unsegmented) actual absolute value of such small numbers.

Large-index block structure: A structure accommodating larger indices, figure 3-2,(c), is required since the 95% of the SYS_i index values are greater than the larger small index. Note that any index generated by the 256-location ATM_i table can fit within a single byte; thus, any burden owing to splitting and joining ATM_i index values is omited since the *Type* field explicitly indicates a 1-byte ATM_I index. We have observed that the compression and encoding/decoding procedures gain in performance and speed due to separate small index blocks instead of a generic index block structure. It has been mentioned that the SYS_i table is built during the initial compilation process and hence, the static table's indices are created in a random order; obviously, the aid in compression of small index blocks can be significantly increased if the first 32 SYS_i indices are explicitly associated with the most frequently referenced symbols such as `car`, `cdr`, `let` etc.

3.7.3 The Low-level FILOS Compression Format

As an example, consider the case the message *msg* displayed below is to be transmitted and evaluated by an RTC. Assume also that the communication channel is clean, that is *msg* is the first transported message, and the RTC has been initialized so that the value of the data sequence `data-seq2` is pointed to by the 9th index of the (remote) HSH_i table. Note that the abbreviations have been fully expanded; for instance, `#'integerp` is expanded to the equivalent expression `(function integerp)` since this expansion is automatically performed during evaluation by the Lisp *reader* prior to any transmission.

msg: `(setq data-seq1 (remove-if-not #'integerp data-seq2))`

The FILOS' encoding procedure generates a series of contiguous variable length blocks, listed in figure 3-3, and some blocks of particular interest are explained in details.

block 2: The built-in special form `setq()` is encoded as a large-index block as follows:

100: Large SYS_i index block (the three most significant header's bits);

00010: the size of the index's absolute value is two bytes; the header is completed.

0000001001100110: The binary value of the SYS_i index 614 (split in 2 bytes).

block 3: The user-defined symbol `data-seq1` is bound to `#<OBJNULL>` in the sender's context and hence it is encoded as a new-object block including the following information:

110: New object;

01010: of type symbol; header is completed.

00001001`data-seq1`: The symbol in the `#<"USER">` package is 9-bytes long; the rest bytes hold the actual characters; a special string termination character is not required.

block 1:	(	*00001000*
block 2:	setq	*000101000000001001100110*
block 3:	data-seq1	*010101100000100*1data-seq1
block 4:	(	*00001000*
block 5:	remove-if-not	*000101000000001011111101*
block 6:	(	*00001000*
block 7:	function	*01111001*
block 8:	integerp	*000101000000000110011000*
block 9:	)	*00011101*
block 10:	data-seq2	*01001010*
block 11:	)	*00011101*
block 12:	)	*00011101*

Figure 3-3: The low-level compression output format.

block 7: The built-in special form `function()` is encoded as a small-index block:

001: Small SYS_i index block.

01111: The SYS_i index 15; the block is completed.

block 10: The user defined symbol `data-seq2` is remotely bound and hence encoded as a small HSH_i index block:

010: Small HSH_i index block.

01001: The HSH_i index value 9; the block is completed.

At a first glance, the example of figure 3-3 reveals the effective compression of messages; the original size of the message *msg* is 62 bytes whereas the compressed one is just 28 bytes; clearly, the 54.83% of the original message size has been excluded form the actual transmitted data quantity without losing or damaging the original message's contents. Moreover, 14.28% of the message size has been physically compressed, due to the removal of four intermediate blanks, and the 85.72% of the size is logically compressed. Another important aspect is the lessening of creation and internment of symbols. For instance, the definition structure of the built-in function `remove-if-not()` is pointed to by th SYS_I index 781 at any network site.

If the same exactly message is transmitted back to the initial sender further compression is feasible; that is, the receiver's encoding process will produce the same series of blocks, as in figure 3-3, but the third block is encoded as *00000000* which means that `data-seq1` is now the ATM_i index zero, denoted by the three most significant bits and the five less significant bits respectively. Consequently, the 70.90% of the initial message size is excluded; this is of vital importance as we will see in section 5.6.1.

3.8 The Encoding/Decoding Algorithm

The encoding/decoding cycle of the FILOS compression algorithm is simple, and its main steps in the case of a sending process can be abridged in the following three procedures:

Step 1: For each symbol check its type field. If it is marked as SYS_i or HSH_i construct the appropriate index block. Else if its index field is non-negative [3] construct an ATM_i index block; otherwise, make a symbol block and associate an ATM_i index with it.

Step 2: For any other object type search the ATM_i table; if an occurrence is found then create an index block of the appropriate size and type; otherwise, encode literally an object block of the appropriate type and associate an ATM_i with it.

Step 3: If the communication buffer runs out of space then encode an *end-of-segment block*, transmit the segment, wait for an acknowledgement message, and then start encoding the rest of the message (or the next segment).

Upon receipt of a message, the remote decoding procedure performs the following two steps:

Step 1: Compare the last byte of the last received byte stream for equality to the *end-of-segment* block; in case of a segment, transmit an acknowledgement system message, start decoding, and wait for more message segment(s); otherwise start decoding.

Step 2: Examine the three most significant bits of the header of each block; if it is an index block then access the appropriate index table and return the associated object; otherwise, create a new object structure, intern it in the encoded package if it is a symbol, and associated an ATM_i index with it.

3.9 Compression Performance

In this section we measure the compression performance extracted when objects of various types and lengths are filtered through FILOS. In all cases the compression results are valid if the following properties are in effect:

- The applied compression test is simple; objects are remotely echoed, i.e. the remote receiving RTC evaluates the message by simply printing it and immediately transmits it back to the initial sender. Thus, the same exactly object travels twice through the same communication channel.

[3] The index and type fileds of a symbol which is not handled by any index table are equal to -1.

- For reasons of convenience compound messages are represented as simple vectors.
- The indentation of messages which enclose programs has been excluded; thus, the compression results refer to a collection of objects separated by single blanks.
- Abbreviations such as the quote sign (*'*) and the function sign (*#'*) have been replaced by their expansions (**quote** *obj*) and (**function** *obj*) respectively.
- Finally, each message is the first one transmitted between the two RTC's preserving a clean environment; that is, the state of the ATM_i table is initialized for each message. Moreover, RTC's are initialized for maximizing the HSH_i table's index benefits.

Consider the case where the message illustrated in figure 3-4 is to be echoed; note that for reasons of an elegant presentation the indentation of the program has been preserved. The original message size is 446 bytes and the following object types are included: (a) 41 symbols of an average print name length seven characters, two strings of total size 35 bytes, four 1-digit fixnums, and 26 lists. The sender's compression procedure results to a message of size 187 bytes or the 41.92% of the initial data size, whilst the receiver's compression procedure results to a further compressed message of size 131 bytes or the 29.37% of the original data size. Clearly, the excluded data quantity is 58.07% and 70.62% of the initial data size before and after the evaluation of the message.

```
#((defvar *hooklevel* 0)

(defun hook (x)
  (let ((*evalhook* (quote eval-hook-function)))
    (eval x)))

(defun eval-hook-function (form &rest env)
  (let ((*hooklevel* (+ *hooklevel* 1)))
    (format *trace-output* "~%~V@TForm:   ~S" (* *hooklevel* 2) form)
    (let ((values (multiple-value-list
                    (evalhook form (function (quote eval-hook-function))
                    nil
                    env))))
      (format *trace-output* "~%~V@TValue:~{ ~S~}" (* *hooklevel* 2) values)
      (values-list values)))))
```

Figure 3-4: The format of an example message.

On the other hand, after the remote decoding of the message only four out of 41 symbols are created and interned corresponding to the 0.97% of the total, three fixnums are created or 100% (the second occurrence of the small fixnum two is literally encoded), and

two string creations or 100%. After the message has been evaluated all of the remotely undefined objects have been replaced by ATM_i indices and therefore, the initial sender's decoding procedure performs no further object creation and internments. Another notable observation is the number of the indices produced by all index tables. The first compression phase generates 23 SYS_i, nine HSH_i and four ATM_i indices, whilst the second phase generates the same number of indices but plus six ATM_i indices. For instance, the symbols `*hooklevel*`, `hook` and `eval-hook-function` are handled as HSH_i indices and their occurrence frequency is five, one, and three times respectively.

This example reveals that the actual transmitted data quantity is 64.34% less than the original one since, the total original data size is 892 (446+446) bytes and the total compressed data size is 318 (187+131) bytes. Similarly, the total number of symbol creations and internments, and fixnum and string creations have been reduced to 95.12%, 62.50%, and 50% respectively.

3.9.1 Special Cases of Compression Performance

It is worthwhile mentioning some special cases of message compression where all of the elements of a list or a simple vector are repetitions of the same object. Consider a simple vector whose dimension is 1000 and whose all elements are the user defined symbol `any-symbol`. For the print name length is 10 characters, the size of the message is 11002 bytes, whilst a bidirectional compression results to a compressed message size of 9.21% (90.78% exclusive data) and 9.16% (90.83% exclusive data) of the original one. In particular, if all elements are quoted the original message size is 19002 bytes since eight additional bytes for each quoted object X are required e.g. $\overset{1\,2\,3\,4\,5\,6\,7\;\;8}{(quote\ X)}$, whilst the excluded data quantity is 78.92% and 78.99% respectively.

3.9.2 Compression Results

Here, we measure the compression performance of FILOS when a number of messages, containing code of real Lisp programs, are bidirectionally transmitted. More precisely, each message consists of objects of different attributes, such as type and size, in order to observe the behaviour of FILOS under numerous circumstances. The final results are emerged by the table 3.1. The columns from left to right denote the original message size in bytes, the total number of built-in and user defined symbols, their average print name length, the compressed message size, the number of object creations and internments produced during

MSG	Original Program Size	Total Num. of Symbols	Average Symbol Length	BEFORE EVAL Program Size	BEFORE EVAL Num. of Interns	AFTER EVAL Program Size	AFTER EVAL Num. of Interns	Total Program Size
P_1	446	41	7.1	187 58.07%	4 95.12%	131 70.62%	0 100%	318 64.39%
P_2	1096	134	5.9	398 63.68%	14 94.77%	294 73.17%	0 100%	692 68.43%
P_3	1363	102	9.2	677 50.33%	16 92.15%	275 79.82%	0 100%	952 65.07%
P_4	2468	214	6.3	1340 45.70%	23 94.62%	578 76.58%	0 100%	1918 61.14%
P_5	2694	333	5.5	1199 55.49%	23 96.54%	947 64.84%	0 100%	2146 60.17%
P_6	5149	622	5.6	1924 62.63%	18 98.55%	1639 68.16%	0 100%	3563 65.40%
SUM	26432	2892	6.6	5725 56.68%	98 96.61%	3864 70.76%	0 100%	9589 63.27%

Table 3.1: Compression results: Sizes are measured in bytes, and the percentages refer to the data quantity excluded.

the compression of the message before and after its evaluation, and finally, the total data quantity transferred. Note that the percentages refer to the amount of the excluded data. The reader notices from table 3.1 that in general the about 65% of the original data quantity of messages is absent from the compressed messages; the evidence suggests that the compression produced may be considered constant, since it is irrelevant from message-dependent factors such as the byte size and the contents.

More specifically, a couple of cases are of particular interest. First, the message P_4 yields a poor compression performance after the first encoding procedure because it includes many different and of long length strings; when all these objects are replaced by ATM_i indices remotely, the performance increases dramatically. Likewise the message P_3 includes many symbols with long print names, about the 70% of the code byte size, and hence the same behaviour is observed, that is an eruption in performance after the second compression. The fact that the same symbols are frequently rereferenced through the program's code, whilst strings are not, yields a better compression than the compression achieved from the message P_4. Second, the phenomenon of the frequent rereference of symbols due to the extended used of recursion in Lisp programs is more apparent in case of the message P_6; this message includes 622 symbols but only 18 of them are unique and hence unknown to the receiver's address space. Finally, vital is also the contribution of FILOS in the way compressed messages are rapidly evaluated upon receipt at their destination; in general, the

first receiver is charged with a negligible overhead due to creating and interning unknown objects, whilst the final receiver (initial sender) is totally discharged even for long messages such as P_6.

3.9.3 Behaviour of the Index Tables

Table 3.2 illustrates the behaviour of the index tables when the messages of table 3.1 are transmitted. The columns from left to right indicate the total number of symbols and the number of indexed objects produced after and before the evaluation respectively. The percentages of the bottom line denote the quotas of each sort of index block in comparison with the total number of their appearance. In all cases the ATM_i indices grow after the evaluation, whilst the SYS_i and HSH_i indices remain constant. This is the further compression achieved by the receiver's encoding procedure, which in figure 3-5,(d) is graphically denoted as the parallel transposition of the (dotted) ATM curve over its previous position. The later is the only stable remark since the behaviour of indices is strongly depending on the types of the objects. The truthlike axiom, the more the SYS_i and HSH_i indices are the more the compression performance is, is not always true. For instance, the message P_2, which yields the best compression result, contains many undefined but multiple repeated objects, whereas the message P_4 encompasses less undefined objects but most of them are different strings.

		BEFORE EVAL			*AFTER EVAL*		
Test	*Symbols*	SYS_i	ATM_i	HSH_i	SYS_i	ATM_i	HSH_i
P_1	41	23	4	9	23	10	9
P_2	134	49	56	15	49	70	15
P_3	102	57	24	1	57	48	1
P_4	214	98	62	16	98	100	16
P_5	333	143	140	27	143	165	27
P_6	622	294	174	127	294	206	127
TOTAL	1446	664	460	195	664	599	195
		50.34%	34.87%	14.78%	45.54%	41.08%	13.37%

Table 3.2: Behaviour of compression indices.

Without loss of generality one can take notice from table 3.2 that 50% of all objects in a Lisp program are built-in primitives. This is very important because the SYS_i indices yield an almost non-existent encoding/decoding overhead. Finally, the anticipation for an inexpensive searching and maintenance of the HSH_i table stated in section 3.5.2 is asserted by the table 3.2.

3.9.4 Persistent Compression via History Indices

It has been mentioned that the size of the ATM_i table is only 256 locations but this size has been proved adequate for serving large scale messages. Strictly speaking, an average 8.59% of the table's capacity is consumed when the messages of table 3.1 are transferred; this is graphically stated in figure 3-5,(c). As a consequence, the possibility for an object, repeated multiple times in the same message, or in a different message being transmitted at a future time, to be treated as an already encoded ATM_i index increases substantially. For instance, consider the following two messages:

msg1: `(print (list 'item1 'item2 'item3))`

msg2: `(reverse (list 'item1 'item2 'item3))`

The filtering of *msg1* results to the (permanent) creation of three ATM_i indices pointing to the equal in number list elements. These pointers are available at a later time when *msg2* is transmitted regarding that other messages may have been transmitted in the mean time and the table has not been cleared. Figure 3-5,(a) shows a graphical comparison between the total number of bytes required by FILOS and SIO (see section 3.10) for the bidirectional transmission of all messages of table 3.1.

3.10 Speed Performance

Although FILOS results to an efficient compression performance it is obvious that an extra overhead due to encoding and decoding objects is required. For that, our effort was focused on the development of a simple algorithm requiring the minimal execution burden (section 3.8), rather than a complex one providing squeezing of more information within single bytes. The following timing results are produced by FILOS and SIO; SIO is a former conventional encoding/decoding mechanism of our system similar to the one provided by the Avalon/Common Lisp [Clamen89] (section 5.5). The main disadvantage of SIO is that only messages of limited size can be transmitted. Moreover, any Lisp object is converted into character strings (default print representation) including blanks and object's special identification characters without attempting any logical or physical compression. Upon receipt the recognition of objects is based on the objects' print representation special characters which are literally encapsulated into the transferred packets; thus, all of the objects are treated as unique and hence, all of them require to be created, looked up and interned in the receiving lexical scope.

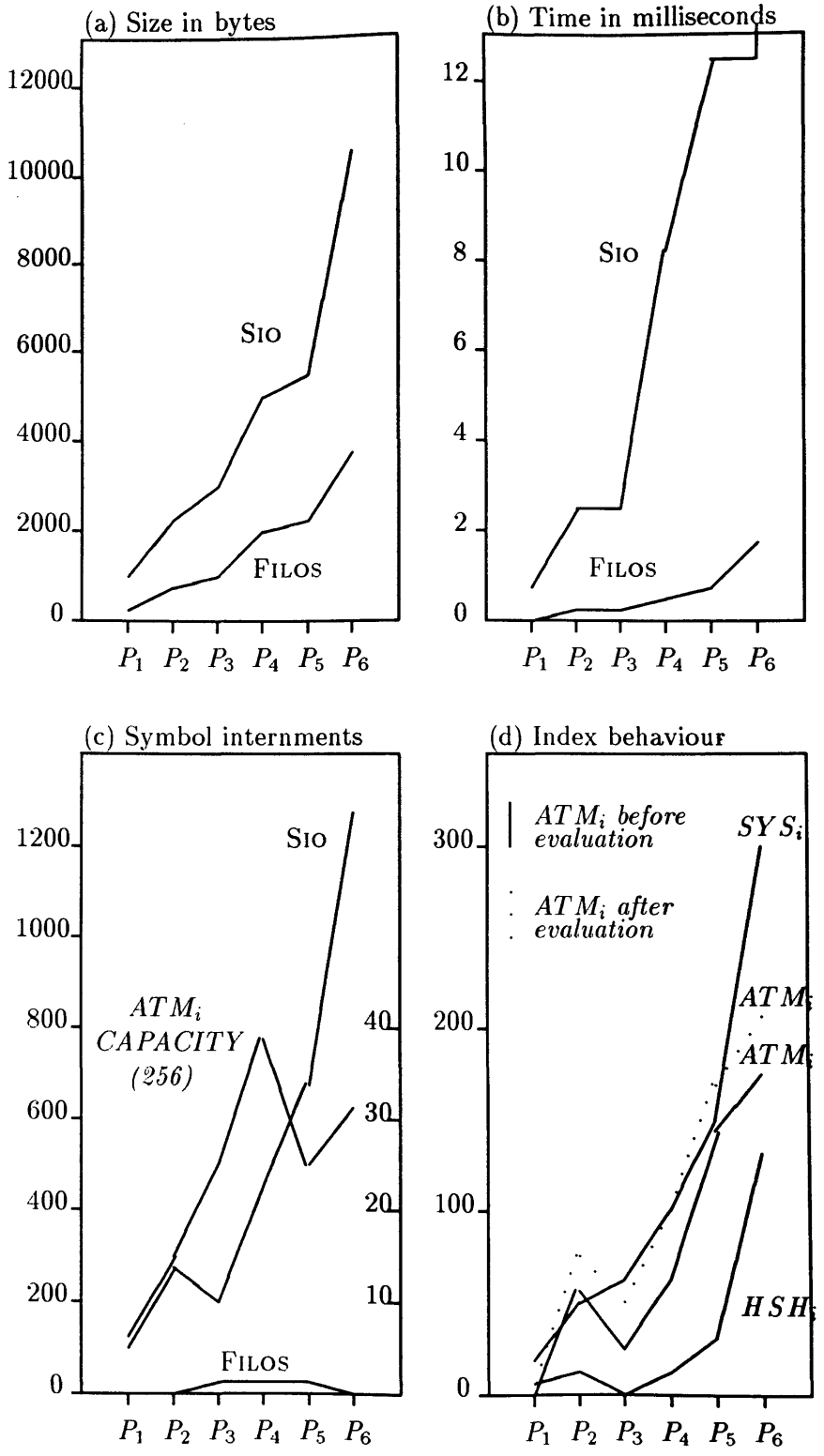


Figure 3-5: LISP program's compression and timing diagrams.

Program	P_1	P_2	P_3	P_4	P_5	P_6
Size	446	1096	1363	2468	2694	5149
SIO	380	772	1526	5389	14741	36960
FILOS	50	117	215	317	933	983
Speedup	7.6	6.6	7.1	17.0	15.8	37.6
Encoding	-	17	67	83	33	67
Decoding	-	17	17	33	50	200
Ratio (i)	-	3.5	2.6	2.8	11.3	3.7
FILOS	17	83	131	233	632	700
Encoding	-	17	50	67	17	50
Decoding	-	-	33	50	33	100
Ratio (ii)	-	4.6	1.6	2.0	12.7	4.7

Table 3.3: FILOS' speed performance (in milliseconds).

It is impractical to do our measurements in the normal way; that is, sending a message, getting the time, receiving the message, getting the time and having the sum as the result transmission time, requires the synchronization of different clocks. Instead, a single clock is used; messages are echoed remotely and the elapsed time [4] is recorded. The procedure is repeated several times and the result is the average transmission time. However, there are some problems in the measurements for several reasons, including the resolution of the clock which is 16.667 milliseconds, the network delays due to the number of the Ethernet's erroneous transmissions, and the differences of the load of the participating workstations. Table 3.3 illustrates the transmission times, in milliseconds, required for echoing all messages listed on table 3.1. The line named **Speedup** states that as the data size increases FILOS requires dramatically less time comparatively to SIO time; clearly, the difference lies in the much slower speed that SIO encodes/decodes objects which is strongly depending on the amount of the transferred bytes. Thus, FILOS is 37.6 times faster in case of the program P_6 which is graphically illustrated in figure 3-5,(b).

Another point of great importance is the speed ratio between the encoding/decoding algorithm and the employed Ethernet. The line named **Ratio (i)** shows that in all cases the encoding/decoding is faster than the network. In particular, when the program P_5 is echoed FILOS is 11.3 times faster than our Ethernet. Note that the UDP/IP protocol has been used for all experiments. Note also that for small data sizes, programs P_1 and P_2, the measured encoding/decoding times were zero; since these values are not correct, the measurements have been hyphenated. Other experiments showed that for relatively short messages (128

[4] The time for our experiments is obtained by `gettimeofday()` (see section 4.3.1 for its definition).

bytes) the measured encoding/decoding time was always zero, but the Ethernet time was, in most of the cases, greater than 30 milliseconds.

The behaviour of FILOS is different when the programs are transmitted multiple times after their first transportation. Due to the extensive use of all categories of indexed objects both FILOS and Ethernet times have been significantly reduced. The line named **Ratio (ii)** states a better FILOS performance over the network's one. However, the decoding phase is more often wasteful in time and the last program is a peculiar example to this case; this problem is discussed in details in section 7.3.

On concluding this compression survey we give an exemplary experiment which highlights the significance of FILOS. In this experiment we transmitted bidirectionally (via UDP/IP) several messages of sizes 384, 1536, and 3072 bytes and the measured mean elapsed times were 0.036, 2.376, and 3.088 seconds. The same messages are retransmitted after the 65% of the original byte size has been excluded (table 3.1); then the measured times were 0.030, 0.474, and 0.283 seconds, i.e. 1.2, 5.0, and 11.0 times faster transmissions. Obviously, this speed up is due to the diminutive burden in segmenting and handling less smaller IP packets (section 1.3); meanwhile, Kimbleton and Schneider [KSSM75] have proved that critical factors such as delay, busy, and idle circuit periods decrease as the transferred data quantities decrease.

3.11 Summary

In this chapter we presented the fundamental concepts of FILOS which yields a highly efficient compression of Lisp data structures; in particular, a series of experiments revealed that the compression performance in general is independent from the messages being manipulated; moreover, the encoding/decoding cost of FILOS also proved minimal in comparison with the transmission cost required by our Ethernet. The benefits of these FILOS' features are of great importance for network-based systems which becomes more apparent in subsequent chapters. In particular, in chapter 4 FILOS constitute the primary factor for efficaciously confronting with various real problems; in chapter 5 the embodying of FILOS in LINDA [NCDG89] entails a significant raise of its overall performance; finally, in chapter 7 we mention several ways FILOS can or could improve its current performance.

Chapter 4

Exploring the Capabilities of Network-Based Systems

4.1 Introduction

HAVING REDUCED DRASTICALLY THE COMMUNICATION OVERHEAD we explore the capabilities of ETHERLISP and hence of network-based systems in general for efficiently confronting with real problems. Distributed algorithms require particular designs different from the ones used for shared-memory algorithms. The primary difference lies on the data dependency and problem's level of complexity which are the essential barriers that usually restrict the performance of a distributed algorithm. Both data and code must be partitioned and dispersed into multiple physically separated address spaces, whilst multiple evaluation results need to be collected and correlated to produce the final problem's solution. These operations are performed via an expensive communications network instead of references to addresses in a shared memory. Clearly, a frivolous design in terms of an improper distribution may cause excessively frequent communications leading to a (mistaken) poor performance.
In this chapter, we present a number of typical problems to point out the cases with which ETHERLISP can or can not cope efficiently. The code of some problems is given for a deeper

understanding of the methodology that we consider suitable for developing compact, scalable, and efficient distributed algorithms.

4.2 Objectives

The objectives being investigated in this chapter can be abridged into the next conjecture:

■ **Conjecture 3** *A sequential algorithm can be thought of as a set of instructions used for describing a real problem into a format executable by a computer. A parallel algorithm furthermore includes a set of conventions specifying the manner real problems and data are decomposed into multiple threads of control that can be scheduled for parallel execution.*
The execution of every sequential algorithm which is determinate, deterministic, and whose both data and processing granularity are coarse-grained or medium-grained, yields time greater than the one required by a parallel algorithm performing on a Multiple-Instruction, Multiple-Data (MIMD) physically distributed configuration. Additionally, a MIMD architecture provides a flexible environment in which alternative ways for achieving efficient, cost-effectiveness, scalable, and totally transparent parallelism are possible.

4.3 Issues on the Characteristics of Distributed Algorithms

Parallel algorithms are strongly dependent on the parallel architecture being executed on. Figure 4-1 shows the processor configuration of our interest. The *Global Control Point (GCP)* is the processor that, through a shared interconnection network, supplies multiple processors with local memories with multiple instruction and data streams. Shortly thereafter, all processors start executing asynchronously, i.e. at their own speeds. Finally, GCP collects all remote results and combines them properly to produce the final solution. Although the MIMD architecture of figure 4-1 is the only one well supported by ETHERLISP, alternative algorithms can be developed when different approaches upon extracting parallelism are used; this relies on the particular concepts and characteristics of parallel algorithms presented next.

Parallelism: The concept of parallelism refers to the reduction of the best sequential execution time using parallel algorithms as they have been defined in conjecture 3. This can

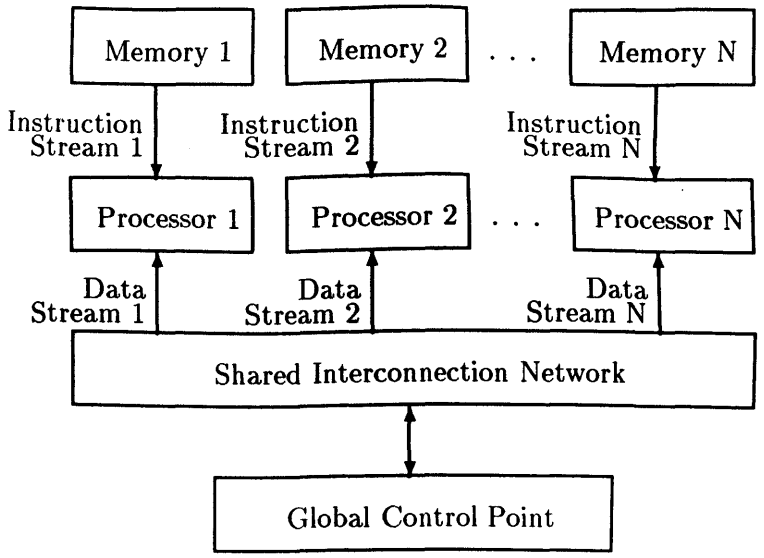


Figure 4-1: Multiple-Instruction, Multiple-Data (MIMD) processor architecture.

be achieved when assigning finer threads of control (processing modules) that manipulate finer data sets (data modules) onto independent processing elements (PE's).

Module granularity: Module granularity deals with the relative size of both data and processing modules. In particular, processing modules are characterized as: (a) *determinate* when the outcome is known given an initial state, (b) *indeterminate* when the outcome is unpredictable from an initial state, (c) *deterministic* when the number of modules spawned at execution time is known given an initial state, and finally, (d) *non-deterministic* when the number of modules spawned at execution time is unpredictable from an initial state. The primary feature that accompanies the module granularity is the overhead required for synchronizing and communicating modules that is critical for network-based systems. Consequently, fine granularity is suitable for shared-memory configurations whereas medium and large granularities suit to loosely coupled PE's. However, physically distributed systems presume a flexible interconnection mechanism that permits rapid point-to-point data transfer when the distance among PE's (long haul configurations) plays important role. ETHERLISP's MIMD environment is independent of the distance among interconnected nodes, and hence arbitrary permutations of source and destination PE's would be possible.

Data dependency: Data dependency is the primary factor for decision making upon designing an algorithm. Major decisions include the specification of patterns for allocating data, e.g. extended use of local memory instead of a network-shared one or vice versa, and consequently patterns for alternative communication schemes, e.g. use of frequent data

transfer when fine granularity is in effect, or a larger-grained organization.

Granularity of parallelism: The granularity of parallelism applied within parallel algorithms can be roughly classified into three categories: (a) *fine-grained*, (b) *medium-grained*, and (c) *coarse-grained* algorithms. The chief difference lies in the granularity of both processing and data modules. The fine-grained class presumes inexpensive creation of a large number of processing modules, as well as simultaneous access to shared data. Strictly speaking, this class is suitable for shared-memory configurations. Oppositely, network-based MIMD architectures are characterized as *exclusive-read, exclusive-write* indicating that no single PE allows simultaneous reading from or writing into the same memory location. GCP repeatedly accepts and processes memory requests encapsulated within messages in a discrete order. Thereby, ETHERLISP is amenable for *divide-and-conquer* oriented techniques that divide problems into smaller ones which actually are instances of the initial problems. This scheme denotes that sequential algorithms can efficiently use recursion, whereas parallel algorithms can proceed in real parallelism.

4.3.1 Notions on Measuring the Performance of Parallel Algorithms

Assuming that P is the number of processors (PE's), t_{seq} the sequential execution time, and t_{par} the parallel execution time, the performance of parallel algorithms, or portions of them, can be measured by the following notions:

•**Speedup**: The speedup S of a parallel algorithm over a sequential one is given by:

$$S = t_{seq}/t_{par}$$

•**Efficiency**: Efficiency E of an algorithm is defined as:

$$E = S/P$$

When $E = 1$, the parallel algorithm is as efficient as the ideal speedup $S^i = t_{seq}/P$.

Next, we present several tests grouped into three case studies exploring fundamental attributes of network-based systems. The outcome of this survey depends on our particular experimental MIMD system and the remarks listed below. However, it is of our belief that this outcome reflects the capabilities of physically distributed systems in general.

(i) All tests ran when the network was light-loaded, e.g. late at night, but part of the computation power was devoted for a large and long-lasting scientific factorization project runing in low priority. Fluctuations proceeded also from the particular network's state since tests were performed sporadically within a wide time period, but individual tests were completed without interruptions within short (usually 24-hours) time intervals.

(ii) The timing results depend on the accuracy attained by the Lisp primitive measuring real elapsed time and whose underlying definition in ANSI C notation is:

```
double get_internal_real_time () {
        static struct timeval begin_tzp;
        struct timeval tzp;
    if (begin_tzp.tv_sec == 0)
        gettimeofday(&begin_tzp, 0);
    gettimeofday(&tzp, 0);
    return((tzp.tv_sec-begin_tzp.tv_sec)*60+((tzp.tv_usec)*60)/1000000);
}
```

All tests have been performed on equivalent Sun's 3/60 workstations but less powerful Suns's 3/50 and 3/75, as some tests revealed, were also used whenever the requirements for a larger number of participating processors arose.

(iii) The sequential algorithms presented are not the best, but the corresponding parallel ones are based on instances of the formers. All individual tests have been ran several times and the mean execution time was recorded and compared with the mean sequential time of all participating processors. Parallel execution times do not include delays for establishing connections among processors.

4.4 Case Study I: Matrix Multiplication

4.4.1 Matrix-by-Matrix Multiplication

The problem of the matrix-by-matrix multiplication is defined as follows. The product of an $m_1 \times n_1$ matrix A and an $m_2 \times n_2$ matrix B is an $m_1 \times m_2$ matrix X whose elements are calculated by $x_{ij} = \sum_{w=1}^{n} a_{iw} b_{wj}$ when $1 \leq i \leq m_1$ and $1 \leq j \leq n_2$. The test assumes that the elements of all matrices are numerals, selected randomly for each test, and the number of rows equals to the number of columns. Figure 4-2 demonstrates a sequential solution that calculates each x_{ij} as the product of the row i and column j. Based on this approach n^2 x's are to be produced, and hence the running time of the algorithm is $O(n^3)$.

The Parallel Algorithm I

The sequential algorithm can be straightforwardly parallelized when, say, half of the problem is given to another processor. The algorithm I of figure 4-3 executes the same steps as the sequential one but any two successive x's are calculated in parallel. Additional power could be provided if the loop's step increments by the number of gradually added PE's P; thus at each iteration P x's are calculated simultaneously. Consequently, the running time is $O(\frac{n^3}{2})$ ($P = 2$ here) which theoretically is faster.

```
(defun seq-mm (matrix1 matrix2)
  (let* ((m1dim (array-dimensions matrix1))
         (rowsize (car m1dim)) (colsize (cadr m1dim))
         (product (make-array (list rowsize colsize)))
         (currow) (curcol))
    (dotimes (i rowsize product)
      (setq currow (get-row matrix1 i rowsize))
      (dotimes (j colsize)
        (setq curcol (get-column matrix2 j colsize))
        (setf (aref product i j) (calc-product currow curcol))))))

(defun calc-product (row col)
  (let ((rlen (length row)) (prod 0))
     (dotimes (i rlen prod) (incf prod (* (aref row i) (aref col i))))))

(defun get-row (matrix rowno rowsize)
  (let ((row (make-array rowsize)))
    (dotimes (i rowsize row) (setf (aref r i) (aref matrix rowno (+ 0 i))))))

(defun get-column (matrix colno colsize)
  (let ((column (make-array colsize)))
    (dotimes (i colsize column) (setf (aref c i) (aref matrix (+ 0 i) colno)))))
```

Figure 4-2: Matrix-by-matrix multiplication: The sequential algorithm.

The Parallel Algorithm II

Apart from the frequent communications in algorithm I, t_{par} nears t_{seq} due to the negligible time required for the calculation of a product (table 4.1). This gives rise to the need for an alternative algorithm illustrated in figure 4-4. The strategy is the reduction of multiple communications to a single transmission when clusters of rows and columns are transferred instead of individual ones. In algorithm II a single cluster encapsulates half of the problem, e.g. half of all rows and columns, independently of the original dimensions of matrices.

```
(defun par-mmI (matrix1 matrix2)
  (let* ((m1dim (array-dimensions matrix1))
         (rowsiz (car m1dim)) (colsiz (cadr m1dim))
         (product (make-array (list rowsiz colsiz)))
         (currow) (cureth (car *ether-list*)))
    (dotimes (i rowsiz product)
      (setq currow (get-row matrix1 i rowsiz))
      (do ((j 0 (+ j 2))) ((= j colsiz)) ; Inner loop (step = 2).
        (let ((loccurcol (get-column matrix2 j colsiz))       ; Current local column.
              (remcurcol (get-column matrix2 (1+ j) colsiz))) ; Current remote column.
          (push-ether (make-msg (neval calc-product) currow remcurcol) cureth)
          (setf (aref product i (1+ j)) (calc-product currow loccurcol))
          (setf (aref product i j) (listen-ether cureth)))))))
```

Figure 4-3: Matrix-by-matrix multiplication: The parallel algorithm I.

Obviously, this design increases parallelism and performance since multiple messages have been replaced by a single cluster, but it implies a rather complex collection mechanism; the result clusters of products computed locally and remotely must be merged and represented as a two dimension matrix (the final result). Theoretically, the running time of the second algorithm is also $O(\frac{n^3}{2})$ since two PE's are used.

```
(defun par-mmII (matrix1 matrix2)
  (let* ((dim (array-dimensions matrix1))
         (rowsiz (car dim)) (clusterrowsiz (/ (car dim) 2)) (clustercolsiz (car dim))
         (remrowcluster (get-row-cluster matrix1 clusterrowsiz 0 rowsiz))
         (remcolcluster (get-column-cluster matrix2 clustercolsiz rowsiz))
         (cureth (car *ether-list*)))
    (push-ether (make-msg (neval calc-cluster-product)    ; The instance.
                          (qeval 'remrowcluster)          ; Half rows.
                          (qeval 'remcolcluster)) cureth) ; Half columns.
    (let* ((offset clusterrowsiz)
           (locrowcluster (get-row-cluster matrix1 clusterrowsiz offset rowsiz))
           (loccolcluster (get-column-cluster matrix2 clustercolsiz rowsiz)))
      (merge-clusters rowsiz (calc-cluster-product locrowcluster loccolcluster)
                             (listen-ether cureth)))))   ; Correlate remote results.

(defun calc-cluster-product (rowcluster colcluster)
  (let (product)
    (dolist (row rowcluster product)
      (dolist (col colcluster)
        (setq product (append product (list (calc-product row col))))))))

(defun get-row-cluster (matrix rowno offset rowsiz)
  (let (rowcluster) (dotimes (i rowno rowcluster)
                      (let ((row (get-row matrix (+ i offset) rowsiz)))
                        (setq rowcluster (append rowcluster (list row)))))))

(defun get-column-cluster (matrix colno colsiz)
  (let (colcluster) (dotimes (i colno colcluster)
                      (let ((column (get-column matrix i colsiz)))
                        (setq colcluster (append colcluster (list column)))))))

(defun merge-clusters (dim cluster1 cluster2)
  (let* ((product (make-array (list dim dim))) (clusterrows (/ dim 2)) (offset (* dim -1)))
    (dotimes (i clusterrows product)
      (incf offset dim)
      (dotimes (j dim)
        (setf (aref product i j) (elt cluster2 (+ j offset)))
        (setf (aref product (+ i clusterrows) j) (elt cluster1 (+ j offset)))))))
```

Figure 4-4: Matrix-by-matrix multiplication: The parallel algorithm II.

4.4.2 Timing Results

Table 4.1 elaborates the matrix-by-matrix problem in terms of comparing the performance achieved by all algorithms. The most important observation is that both parallel algorithms

are faster than the sequential one, while algorithm II is faster than I as the fifth column ($S^{\star}$) indicates. For instance, in case of 28×28 matrices I's t_{par} is 10.9% faster than t_{seq} and II's t_{par} is 59.0% faster than t_{seq}, whilst II's t_{par} is 48.1% faster than I's t_{par}. Important is also that algorithm II appears to be faster than the ideal speedup (sixth column), as well as inexpensive in resources since $E > 1$ for all cases (see section 4.7 for of these unexpected results interpretation). Remarkable is the weakening in performance of algorithm I for medium size dimensions, whereas the performance increases as the problem's size increases. One possible explanation might be the random state of the network at that time. However, the performance of both parallel algorithms fluctuates for large matrices but steadily follows a descending course.

4.4.3 Vector-by-Matrix Multiplication

The vector-by-matrix multiplication problem is also examined to intimate the importance of reducing communications, as well as an alternative manner of achieving it. The problem was solved utilizing the sequential and the parallel algorithm I used for the matrix-by-matrix problem after they have been properly altered. Clearly, the vector is a matrix with a single row and therefore the encapsulation of the same row within every message for every product being produced remotely is redundant. A second parallel algorithm overcomes this problem by binding the vector remotely before entering the main loop. Thus, each message contains a one-byte small ATM_i index, which represents the raw print name of the vector (row), and a fresh literal column specified by the current loop's counter value. Note that the print name of the symbol representing the vector remotely has been replaced by an ATM_i small index after `setr()` (see section 2.11.2) has been applied. Upon receipt, the index block is replaced by its actual value when it is passed as an argument to the function `calc-product()`. Table 4.2 shows the further speedup $S^{\star}$ attained due to this technique. This should be expected since each message of the second parallel algorithm carries approximately 50% less data than each corresponding message of the algorithm I. However, it is apparent that although the problem is solved faster by both parallel algorithms the very low problem's complexity leads to a poor speedup, whilst parallelism is costly in resources (E^{II} subject to the comments in section 4.7). Moreover, the descending course of the performance is discernible from early stages.

Size	t_{seq}	t_{par} par-mmI		par-mmII		$S^{\star}$	S^{II}	E^{II}
4x4	0.84	0.44	47.6%	0.20	76.2%	+28.6%	4.2	2.1
8x8	2.16	1.64	24.1%	0.82	62.1%	+38.0%	2.6	1.3
12x12	5.65	5.53	2.1%	1.97	65.1%	+63.0%	2.7	1.4
16x16	11.36	11.02	3.0%	4.87	57.1%	+54.1%	2.3	1.2
20x20	21.85	20.66	5.5%	8.53	61.0%	+55.5%	2.6	1.3
24x24	42.61	35.58	16.5%	15.21	64.3%	+47.8%	2.8	1.4
28x28	61.11	54.45	10.9%	25.08	59.0%	+48.1%	2.4	1.2
32x32	95.17	79.20	16.8%	36.90	61.2%	+44.4%	2.6	1.3

Table 4.1: Matrix-by-matrix multiplication: Comparative results between algorithms I and II when $P = 2$, speedup expressed in percentages (%), $S^{\star}$ the additional speedup in % of algorithm II over I, S^{II} and E^{II} II's speedup and efficiency, and time measured in seconds.

Size	t_{seq}	t_{par} *Algorithm I*		*Algorithm II*		$S^{\star}$	S^{II}	E^{II}
1x4	0.152	0.091	40.1%	0.066	56.6%	+16.5%	2.3	1.2
1x8	0.473	0.197	58.4%	0.178	62.4%	+4.0%	2.7	1.4
1x12	0.455	0.395	13.2%	0.354	22.2%	+9.0%	1.3	0.7
1x16	1.156	0.600	48.1%	0.555	52.0%	+3.9%	2.1	1.1
1x20	1.121	1.022	8.8%	0.827	26.3%	+17.5%	1.4	0.7
1x24	1.694	1.185	30.1%	1.159	31.6%	+1.5%	1.5	0.8
1x28	2.370	1.706	28.0%	1.540	35.1%	+7.1%	1.5	0.8
1x32	3.217	2.244	30.3%	1.970	38.8%	+8.5%	1.6	0.8

Table 4.2: Vector-by-matrix multiplication: Comparative results between algorithms I and II when $P = 2$, speedup expressed in percentages (%), $S^{\star}$ the additional speedup in % of algorithm II over I, S^{II} and E^{II} II's speedup and efficiency, and time measured in seconds.

4.5 Case Study II: Searching

As a second case study the problem of searching is considered. Given two unordered numeral sequences $P = \{p_1, p_2, ..., p_n\}$ and $S = \{s_1, s_2, ..., s_n\}$, it is required to determine a sequence $X = \{x_1, x_2, ..., x_m\}$ which contains all $s_i \in S$ that match in value any elements (patterns) in P. Assuming n is P's and S's length valid outputs might be none, n, or any number of matching elements since duplicates are also included. The test consists of three parts each reflecting a different level of processing granularity; thus, 1, $\frac{n}{2}$, and n elements of P are searched for equality with n elements in S. Figure 4-5 illustrates the generic sequential algorithm whose running time for each case is $O(n)$, $O(\frac{n}{2} \times n)$, and $O(n \times n)$ respectively.

```
(defun seq-search (patterns sequence)
  (let (result)
    (dolist (p patterns result)
      (dolist (s sequence) (when (= p s) (setq result (append result (list s))))))))
```

Figure 4-5: Searching: The sequential algorithm.

The Parallel Algorithm

Another interesting attribute of distributed systems is their behaviour when multiple processors co-operate for a single goal. In the light of this issue, a generic parallel algorithm that utilizes more than two processors was developed as illustrated in figure 4-6. The fundamental feature of generic distributed algorithms is the totally transparent manner data should be evenly dispersed and evaluation results should be efficiently collected. In particular, `net-partition()` divides the data set of the problem into $P + 1$ equal portions which are broadcast to P processors (RTC's). After a data portion has been received - the algorithm does not acknowledge transmissions - a previously allocated instance of the sequential algorithm (or a part of it in other cases) accepts it as a parameter. Consequently, all processors, including GCP, start executing at their own speeds. When GCP has finished computing it starts collecting results. Parallelism increases since a portion of the entire problem has been completed locally (no network requirements), and any potentially available remote result(s) are collected and correlated (appended in the example) while other processors might be still in progress. Note that a more genuine approach could burden GCP (root) with less calculation overhead for the sake of an eager collection process. Vital is also the fact that generic parallel algorithms in ETHERLISP provide high levels of scalability. Any algorithm designed as the one in figure 4-6 after it has been debugged, tested, and run for two processors then it safely runs for any number of processors. The theoretical parallel running time is $O(n/P)$, $O((\frac{n}{2} \times n)/P)$ and $O((n \times n)/P)$ for each test's case respectively.

4.5.1 Timing Results

Table 4.3 shows the total searching results of the three problem's cases. In the first case, the low complexity of the searching problem in comparison with a single element being searched, results to a slower execution of the parallel algorithm over the sequential one. Things are getting better when the problem becomes more complex as the second and third cases show. Despite the additional computation power provided by multiple processors the

```
(defun par-search (p s) (net-partition p s))

(defun net-partition (p s)
  (let* ((ssiz (/ (length s) (1+ *number-of-ethers*))) ; Number of S's portions.
         (soffset (- 0 ssiz)))                         ; Start of next S's portion.
    (dolist (eth *ether-list*)
      (push-ether (make-msg (neval seq-search)         ; The instance.
                            (qeval 'p)                 ; Whole P and a portion of S.
                            (qeval '(partition s ssiz (incf soffset ssiz))) 0) eth))
    (append (seq-search p s (1+ ssiz)) (collect *number-of-ethers*))))

(defun partition (s ssiz soffset) ; Partition S.
  (let (segment)
    (dotimes (i ssiz segment)
      (setq segment (append segment (list (elt s (+ i soffset))))))))

(defun collect (n) ; Collect and correlate remote results.
  (cond ((= n 1) (listen-ether (car *ether-list*)))
        ((= n 2) (append (listen-ether (select-ether)) (listen-ether (select-ether))))
        (t (append (listen-ether (select-ether)) (collect (1- n))))))
```

Figure 4-6: Searching: The parallel algorithm.

best performance in speedup and efficiency is attained when $P = 2$. In general, for all cases but the first one the applied parallelism entails faster execution times but the performance follows a steady and fast descending course. This is primarily caused due to two reasons. First, the byte size of the remote results is unpredictable in terms of transmitting either none or n elements when none or n matching elements are found. This is true since each test's repetition is supplied with randomly generated data, fact that implies an imbalanced communication and FILOS's encoding/decoding overhead especially for large data volumes. Second, the generic parallel algorithms are in general vulnerable under heavy network traffic (see also section 4.7). It is crucial that generic algorithms yield their best performance only when all participating processors start with the smallest possible broadcast delays. It should be noticed that in case of short remote execution times (very fine processing granularities) small network delays might be desirable. Assume for example that three processes start at times t_i, t_{i+1}, and t_{i+2} respectively; it is highly probable that GCP will collect the remote results in the same order as the processes started while all system's resources (including the network) are in progress. More precisely, GCP will be in a spin-lock state until the arrival of the first result; the two successive (and discrete in time) inspections for pending messages on the communication channels will yield to the immediate (eager) acceptance of the remaining two results; or alternatively interpreted, the wasted time that GCP is in a (usless) spin-lock state awaiting subsequent messages becomes negligible.

Size	t_{seq}	t_{par} 2	3	4	5	6
1:32	0.072	0.061	0.105	0.149	0.122	0.220
		15.2%	-31.4%	-48.2%	-40.9%	-67.2%
1:64	0.175	0.141	0.191	0.200	0.261	0.239
		19.5%	-8.3%	-12.5%	-32.9%	-26.7%
1:128	0.430	0.308	0.362	0.408	0.404	0.433
		28.4%	15.9%	5.2%	6.1%	-0.7%
1:256	0.728	0.950	1.167	0.883	0.908	0.933
		-23.3%	-37.6%	-17.5%	-19.8%	-21.8%
1:512	2.209	1.167	3.691	3.810	3.616	3.983
		47.2%	-40.1%	-42.0%	-38.9%	-44.5%
16:32	1.016	0.450	0.530	0.591	0.689	0.712
		55.8%	47.9%	41.8%	32.2%	30.0%
32:64	4.846	1.737	2.333	2.549	2.983	2.665
		64.2%	51.9%	47.4%	38.5%	45.0%
64:128	17.750	8.350	10.625	12.074	12.417	12.949
		53.0%	40.0%	32.1%	30.2%	27.1%
128:256	97.899	40.917	52.817	59.075	63.033	63.517
		58.3%	46.1%	40.0%	35.7%	35.2%
256:512	560.229	235.017	301.967	342.611	352.050	364.567
		58.1%	46.1%	39.1%	37.2%	35.0%
32:32	2.210	0.867	1.158	1.233	1.217	1.300
		60.7%	47.7%	44.3%	45.0%	41.2%
64:64	9.409	4.308	4.700	5.643	5.944	5.852
		54.3%	50.1%	40.1%	36.9%	38.1%
128:128	41.397	16.683	21.401	24.567	25.074	25.933
		60.0%	48.4%	41.0%	39.5%	37.4%
256:256	201.497	83.667	112.067	119.783	126.483	129.917
		58.5%	44.4%	41.0%	37.3%	35.6%
512:512	1423.615	514.200	688.800	780.950	882.850	852.967
		64.0%	51.7%	45.2%	38.0%	40.1%

Table 4.3: Searching $1, \frac{n}{2}, n$-length random numerals in a n-length random sequence when $P = 1, 2, ..., 6$, speedup is expressed in percentages, and time is measured in seconds.

4.6 Case Study III: Sorting

The problem of sorting is stated as follows; given a sequence $S = \{s_1, s_2, ..., s_n\}$ whose elements are random numerals, it is required to determine a sequence $S' = \{s_1', s_2', ..., s_n'\}$ such that $s_i' < s_{i+1}'$, $\forall i = 1, 2, ..., n-1$. For reasons being explained shortly afterwards, the trivial recursive quicksort algorithm showed in figure 4-7 is chosen for solving the problem sequentially. A typical optimal running time of quicksort in the average case is $O(n \log n)$.

```
(defun seq-qs (vec) (quick-sort 0 (1- (length vec)) vec) vec)

(defun quick-sort (left-edge right-edge vec)
  (let (last-position)
    (when (>= left-edge right-edge) (return-from quick-sort vec))
    (swap vec left-edge (round (/ (+ left-edge right-edge) 2)))
    (setf last-position left-edge)
    (do ((i (1+ left-edge) (1+ i)) (n right-edge)) ((> i n))
      (when (< (aref vec i) (aref vec left-edge))
            (incf last-position) (swap vec last-position i)))
    (swap vec left-edge last-position)
    (quick-sort left-edge (1- last-position) vec)     ; Sort the first part.
    (quick-sort (1+ last-position) right-edge vec))) ; Sort the second part.

(defun swap (vec i-elt j-elt)
  (let ((temp (aref vec i-elt)))
    (setf (aref vec i-elt) (aref vec j-elt))
    (setf (aref vec j-elt) temp)))
```

Figure 4-7: Quicksort: The sequential algorithm.

The Parallel Algorithm

It is generally acceptable that the problem of sorting is a very important example that puts under trial the capabilities of the features provided by both software and hardware organizations. On the approach of sorting when fine-grained granularity is in effect both shared-memory and MIMD network parallel architectures are forced to extensively use critical resources. Shared-memory approaches might impose a large number of processes to be spawned at run-time; as an effect substantial delays are caused due to synchronizing multiple threads of control. Indicatively, Yuen[Yuen90] mentions that 10231 *futures* (processes) are created when a sequence of 400 random integers is sorted in MultiLisp[Halst85]. As the time for creating a *future* process on the MC68000 processor is 6.5 milliseconds (1.3 to 1.6 milliseconds for a function call) the 79% of the total algorithm's running time is occupied for process manipulation. On the other hand, a MIMD network fine-grained approach impose extensive use of the costly network since element comparisons are performed through multiple transportations among physically disjointed address spaces. Alternatively, by relying on the recursive attribute of the algorithm of figure 4-7 for developing a parallel one [Gehani89b] is again inefficient because such a scheme also suffers from frequent communications and expensive process handling. Divide-and-conquer (figure 4-8) seems to be again the most convenient approach. However, the tight data dependency of sorting involves large overheads for correlating results; the later are sorted irrelevant subsequences

requiring merging that in the worse case results to an $O(n)$ optimal running time; hence, the total parallel running time is faster and equal to $O(\frac{n \log n}{P})$.

```
(defun par-qs (vec) (net-partition vec))

(defun net-partition (vec)
  (let* ((seg-size (/ (length vec) (1+ *number-of-ethers*)))
         (seg-start (- 0 seg-size)))
    (dolist (eth *ether-list*)
      (push-ether (make-msg (neval quick-sort) 0 (1- seg-size)
                            (partition vec seg-size (incf seg-start seg-size))) eth))
    (quick-merge (quick-sort 0 (1- seg-size) (partition vec seg-size
                                                        (incf seg-start seg-size)))
                 (collect-results *number-of-ethers*))))

(defun partition (vec seg-size seg-start)
  (let ((segment (make-array seg-size)))
    (dotimes (i seg-size segment)
      (setf (aref segment i) (aref vec (+ i seg-start))))))

(defun collect-results (res-num)
  (cond ((= res-num 1) (listen-ether (car *ether-list*)))
        ((= res-num 2) (quick-merge (listen-ether (select-ether))
                                    (listen-ether (select-ether))))
        (t (quick-merge (listen-ether (select-ether)) (collect-results (1- res-num))))))

(defun quick-merge (seq1 seq2)
  (let ((len1 (length seq1)) (len2 (length seq2)))
    (do ((newseq (make-array (+ len1 len2)))
         (j 0 (1+ j))
         (i1 0) (i2 0))
        ((and (= i1 len1) (= i2 len2)) newseq)
      (cond ((and (< i1 len1) (< i2 len2))
             (cond ((< (elt seq1 i1) (elt seq2 i2))
                    (setf (elt newseq j) (elt seq1 i1)) (incf i1))
                   ((< (elt seq2 i2) (elt seq1 i1))
                    (setf (elt newseq j) (elt seq2 i2)) (incf i2))
                   (t (setf (elt newseq j) (elt seq1 i1)) (incf i1))))
            ((< i1 len1) (setf (elt newseq j) (elt seq1 i1)) (incf i1))
            (t (setf (elt newseq j) (elt seq2 i2)) (incf i2))))))
```

Figure 4-8: Quicksort: The parallel algorithm.

4.6.1 Timing Results

Despite the costly merging table 4.4 reveals a dramatic increase in performance especially for large sequence sizes and number of processors. A weakening in performance is observed for $P = 4$ perhaps because the reduced complexity of the problem can not cover both communications and merging costs. However, when $P = 10$ the algorithm starts bowing slightly fact that is more apparent for short sequences, since for large ones the algorithm

Size	t_{seq}	t_{par}								
		2	3	4	5	6	7	8	9	10
16	1.1	0.3	0.3	0.7	0.5	0.6	0.6	0.7	0.7	0.8
		70.0%	71.3%	65.4%	**72.8%**	66.1%	43.0%	25.3%	39.1%	**63.0%**
32	1.2	0.6	0.4	0.7	0.5	0.6	0.6	0.7	0.7	0.8
		51.5%	64.6%	39.9%	**70.0%**	54.3%	53.4%	45.2%	44.1%	**36.0%**
64	2.0	0.9	0.8	1.0	0.9	1.1	1.5	1.6	1.6	1.3
		54.6%	61.0%	48.7%	**52.5%**	42.9%	25.7%	21.0%	19.3%	**35.0%**
128	4.8	2.6	1.8	1.6	2.1	2.0	2.2	2.3	2.5	3.1
		46.8%	61.6%	65.3%	**55.6%**	57.6%	52.9%	52.3%	46.2%	**34.3%**
256	12.9	5.2	4.4	4.3	4.0	4.2	4.4	5.1	5.4	6.1
		59.6%	65.7%	66.4%	**68.9%**	67.3%	65.7%	60.2%	57.9%	**52.2%**
512	36.8	13.1	9.6	8.5	8.8	9.2	9.7	10.3	11.0	11.6
		64.3%	73.8%	76.9%	**76.3%**	75.0%	73.8%	72.1%	70.3%	**68.6%**
1024	124.0	40.8	24.0	20.7	19.7	21.3	20.9	22.5	25.3	25.3
		67.2%	80.7%	83.3%	**84.1%**	82.8%	83.2%	81.9%	79.6%	**80.0%**
2048	408.5	114.6	70.0	50.7	45.4	43.0	44.8	47.6	48.7	50.2
		72.0%	83.0%	87.6%	**88.9%**	89.5%	89.1%	88.4%	88.1%	**87.7%**
3072	915.1	233.0	125.4	91.2	79.0	70.6	70.3	77.2	75.5	77.4
		74.6%	86.3%	90.1%	**91.4%**	92.3%	92.3%	91.6%	91.6%	**91.5%**
4096	1630.8	248.2	201.7	143.1	126.4	104.3	97.2	104.6	105.4	110.5
		84.8%	87.8%	91.2%	**92.3%**	93.6%	94.1%	93.6%	93.5%	**93.2%**

Table 4.4: Quicksorting integers when $P = 1, 2, .., 10$, speedup is expressed in percentages, and time is rounded up to one decimal and measured in seconds.

still yields an average 90% faster times. This observation is moderated by the following remarks. First, the performance of the algorithm for short sequences definitely depends somewhat on the (random) ordering of the particular sequence being sorted in each iteration of the test. Second, network delays play a crucial role compared with the low complexity (relatively short execution time) of the problem. Obviously, the algorithm is influenced insignificantly by these factors for large sequences. Despite these effects, we can not explain the extraordinary result for $P = 7$ and a sequence of 4096 elements that has a speedup of 17 times. We discuss this in depth in section 4.7. However, the evidence suggests that both algorithm's and ETHERLISP's structures efficiently handles the critical resources (network and (many) processors) that usually restrict the extraction of efficient network-based parallelism. It should be recalled that seven of the processors were Sun's 3/60 while the rest were less powerful Sun's 3/50.

4.6.2 Extracting Parallelism when Data Dependency is Loose

A common feature to all problems encountered so far is the overhead owing to correlating remote results. This overhead is negligible when a problem consists of multiple logically independent subinstances because the correlation burden degrades to a simple collection of

results. Though data dependency can be of any granularity within single subinstances the absence of relationship among them indicates that the overall data dependency is absent or very loose. The problem of sorting (coarse granularity) has been slightly altered to meet this property. The elements of the sequence being sorted are equal in length randomly constructed numeral sequences, and hence the desirable final result is the sorted independent elements (sequences).

Table 4.5 states the expected results. By corresponding n elements (e's) to n processors (p's) the performance exceeds too often the the ideal speedup (see section 4.7). The algorithm behaved similarly when more processors were added providing that each element is assigned to a single (and dedicated) processing element.

	t_{seq}			t_{par}			S		
Size	$e=2$	$e=3$	$e=4$	$p=2$	$p=3$	$p=4$	$p=2$	$p=3$	$p=4$
32	3.333	2.700	3.350	1.383	1.558	0.800	2.409	1.732	4.187
64	4.138	6.144	8.314	2.100	2.049	1.583	1.970	2.998	5.252
128	9.377	17.377	19.438	4.208	4.682	4.016	2.228	3.711	4.840
256	21.744	32.649	44.156	10.208	10.358	9.324	2.130	3.152	4.735
512	54.444	70.066	95.216	21.783	22.441	23.691	2.513	3.122	4.019
1024	107.316	162.183	207.625	50.241	48.833	50.667	2.136	3.321	4.090

Table 4.5: Sorting independent numeral sequences, when $P = 1, 2, ..., 4$ and time is measured in seconds.

4.6.3 FILOS's Effects on the Performance of Parallel Algorithms

In chapter 3 FILOS was presented as a mechanism that efficaciously compresses and encodes/decodes data structures. To assert it in real terms the problem of sorting is considered when data sequences consist of symbols and strings extracted from real Lisp programs. Table 4.6 shows the timing results compared with the ones produced when data were equal in length 1-digit integer sequences. It should be noticed that FILOS creates a 3-byte block for each (small) integer, and that sorting symbols entails larger t_{seq}'s because the ordering criterion is more costly than simple integer comparisons. The table emerges a better performance for symbols even when the data byte size is 60% to 70% more than in case I. As the difference in the transferred byte size approaches the half, case I gains in performance; this implies a weakness due to encoding/decoding large sequences but the algorithm is still (unreasonably) efficient and cost-effective (section 4.7). Note that all repetitions but the first of a single test make extensive use of indices as the same (but unsorted) sequence is

applied to each repetition (see section 3.9.5).

Size	*Integers (I)*			*Symbols (II)*		
	Bytes	t_{seq}	t_{par}	*Bytes*	t_{seq}	t_{par}
44	264	1.6	0.8 49.9%	662 60.1%	1.9	0.7 63.8%
104	624	3.2	1.8 44.8%	2142 70.9%	4.7	2.6 45.0%
128	768	4.6	2.2 51.3%	1588 48.4%	5.7	3.0 46.5%
576	3456	41.2	15.4 62.5%	6786 49.1%	47.1	21.3 54.8%
808	4848	74.1	26.0 64.9%	10516 53.9%	65.9	29.6 55.1%

Table 4.6: Comparative results when quicksorting n one-digit integers (I) and n symbols and strings (II) when $P = 2$, and time measured in seconds.

4.7 Discussion

The survey of the above three characteristic case studies indicates the fundamental factors that determine the performance of distributed algorithms, and therefore the performance of network-based systems. These factors include the complexity, communications, and data dependency costs of a problem. Figure 4-9 clearly illustrates that problems with coarse-grained complexity (sorting) yield to an enormous performance clearly because costs due to communications and data dependency are supplanted from the large diminution in computing when the problem is segmented into (simplified by) multiple subinstances of very fine-grained complexity. In particular, in the matrix multiplication problem the parallel algorithm is sensitive (weak) for medium size matrices while for large ones it still yields a satisfying performance. Similarly the poor performance of the searching algorithm increases when the algorithm is designed as a generic one (when $P > 2$) since the fine-grained complexity is altered to medium-grained. In general lines, when the complexity of a problem can be characterized (or converted) as medium-grained then it is highly probable that efficient network-based paralellism can be extracted.

The important role of the complexity factor becomes more apparent in the case of the two last case studies. Searching elucidates that MIMD network architectures are inferior in performance than shared-memory configurations in case of fine-grained complexity problems. Despite the propitious low communications, loose data dependency, and large number of processors, both speedup and efficiency are very poor in depth and duration as is graphically illustrated (4-9,ii). Conversely, the coarse-grained sorting has been encountered with a facsimile algorithm but the performance is excellent (4-9,iv). However, the observed super-

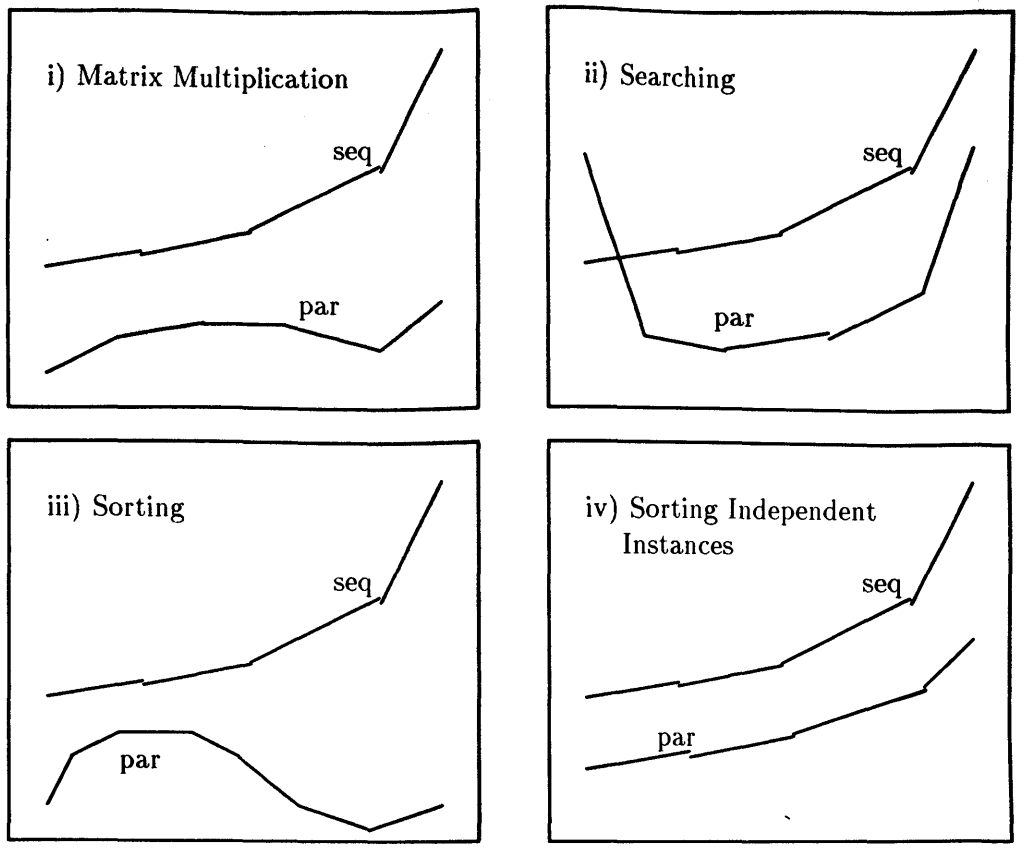


Figure 4-9: Graphical representation of the case studies.

linear speedup, attained also by the other outlined algorithms, might be due to several reasons, including experimental errors, clock accuracy, difficulties in timing remote events, and the randomly selected data sequence for each repetition of a test (though we repeated the tests several times and these effects should average away). But the most reasonable explanation is that the algorithmic complexity is changed by the parallel algorithms. In case of sorting the sequential running time is $O(n^2)$ in the worst case (which may have been evoked by the form of the test data), but divide and conquer transforms the parallel running time to $O(nlogn)$. This has as an effect a faster parallel algorithm even if the later runs on a single processor. For instance, by sorting numeral sequences of 512, 1024 and 2048 elements by the algorithms in figures 4-7 and 4-8 on a single processor the measured sequential execution times were 3.2, 9.0 and 33.0 seconds[1] whist the parallel ones when $P = 4$ were 1.0, 2.75 and 14.0 seconds respectively; thus, the measured speedup (t_{seq}/t_{par}) is 3.2, 3.272 and 2.357 for each case. Moreover, Bradford and Padget in[Brad93] observe

[1]This experiment has been performed on a 29 MIPS machine where ETHERLISP has been recently ported.

that a super-linear speedup is possible when divide and conquer is applied on a bubble sort algorithm. These figures demonstrate the general difficulties in comparing sequential algorithms with parallel ones, but despite the anomalous results, we believe the underlying speedup due to parallelism (and not due to algorithmic effects) is true and useful.

Table 4.7 presents the timing results of the critical factors (portions) of both searching and sorting algorithms. According to the order of their appearance on the table the factors denote the data segmentation and distribution cost, the execution time of a subinstance, the cost of correlating results, and the bidirectional data transportation (via UDP/IP) cost. When $P = 2$ and both algorithms operate on 64-element integer sequences (same data transportation requirements) the t_{udp} cost is about 0.05 seconds for both algorithms. But comparatively to the total execution time of the algorithm the searching's t_{udp} cost corresponds to the 33.7% whereas the sorting's equivalent cost corresponds to the 3.3%. The same is observed when $P = 4$. Finally, the influence of the data dependency factor on the performance of an algorithm is specified by the corresponding t_{corr} costs. The absence of tight data dependency is interpreted as an increased t_{exec} in case of searching, whereas in the sorting case t_{corr} causes a significant overhead.

	SEARCHING				*SORTING*			
	$P = 2$		$P = 4$		$P = 2$		$P = 4$	
Cost	1:64	256:256	1:64	256:256	64	1024	64	1024
t_{dist}	33.7%	1.5%	31.7%	0.4%	3.3%	3.0%	5.0%	6.6%
t_{exec}	29.1%	98.3%	23.7%	99.3%	57.3%	82.7%	25.8%	47.4%
t_{corr}	10.0%	0.2%	5.1%	0.2%	10.7%	3.7%	8.0%	28.1%
t_{udp}	12.3%	0.1%	15.3%	0.2%	2.9%	0.8%	22.6%	5.2%

Table 4.7: Comparing the critical features of distributed algorithms when, t_{dist} is the data distribution cost, t_{exec} is the execution time of a subinstance, t_{corr} is the data correlation cost, t_{udp} is the UDP/IP cost, and P is the number of PE's.

Apart from the particular case studies several other important relevant to the subject observations influence the performance of network-based algorithms:

Timing fluctuations: The mean execution time of all tests was not calculated automatically but we rather let each iteration reporting its elapsed time. In this way, we observed that the fluctuations between successive sequential measurements were insignificant. On the contrary, parallel measurements fluctuated significantly denoting that the state of a network changes within short time intervals. In particular, the first measurement often yielded time double than the rest repetitions of a test for time intervals less than three or five seconds.

This denotes that the network entered in a "frozen" state while the initialization, **e.g.** supply with fresh data set(s), of the next test's repetition was taking place.

Hosts' load effects: The initiating processor (LTC) hosting GCP was completely devoted on executing each particular test; that was not always true for the rest participating processors (with potentially active users). We observed that the execution time of any test was approaching the double one whenever time consuming processes, e.g. preparing documents with *Latex* or compiling programs, were performing in the background.

UDP/IP vs. TCP/IP: In all case studies, and during this research in general, UDP/IP was preferred for interconnecting processors. Though unreliable it has been proved as highly reliable regardless the frequency and byte size of the messages interchanged. UDP/IP also yielded faster transmission times in comparison with TCP/IP. This was the outcome of two experiments performed at widely distinct time periods.

Test	*Bytes*	*TCP*	*UDP*	*UDP Speedup*	
P_1	318	189	44	4.295	76.72%
P_2	692	255	66	3.863	74.11%
P_3	952	372	122	3.049	67.20%
P_4	1918	939	255	3.682	72.84%
P_5	2146	622	266	2.338	57.23%
P_6	3563	1122	771	1.455	31.28%

Table 4.8: Timing TCP/IP and UDP/IP transmission speeds (time in milliseconds).

		TCP		*UDP*			
Size	*Bytes*	t_{par}	$D_{\overline{x}}$	t_{par}	$D_{\overline{x}}$	*UDP Speedup*	
16	102	0.573	1.857	0.273	0.153	2.098	52.4%
32	198	1.573	6.569	0.476	0.123	3.304	70.3%
64	390	1.497	3.807	1.395	2.761	1.073	8.8%
128	774	2.513	3.137	2.319	2.118	1.083	7.7%
256	1542	5.903	6.659	5.066	1.202	1.165	14.2%
512	3078	10.150	1.734	10.166	2.000	0.998	-0.2%
1024	6150	24.866	1.067	24.458	1.682	1.016	1.6%
2048	12294	60.216	0.600	62.291	5.016	0.996	-3.4%
3072	18438	110.525	0.884	110.608	1.283	0.999	-0.1%
4096	24582	173.967	0.227	170.083	0.184	1.022	2.2%

Table 4.9: Comparative results between TCP/IP and UDP/IP endpoints when quicksorting integers, $P = 4$, $D_{\overline{x}} = \sum_{i=0}^{n-1} \mid \overline{X} - x_i \mid$, and time measured in seconds.

Table 4.8 demonstrates the transmission speed of both protocols when a series of Lisp programs are bidirectionally transported. Apparently, UDP/IP is amenable for rapidly

transmitting small and medium data volumes. Similarly, table 4.9 illustrates the timing measurements when numeral sequences of various lengths are quicksorted. Again, the same behaviour is observed. Furthermore, TCP/IP causes long delays denoted by the standard deviation from the mean test's execution time ($D_{\overline{x}}$), whereas UDP/IP performs in a smoother manner. This is mainly due to short test's running time and the significant delays owing to acknowledging underlying TCP/IP data segments. Finally, both protocols perform similarly for large data volumes.

4.8 Summary

The conjecture stated early in this chapter has been fully proved because we believe that the case studies have been chosen in such a way that cover a broad range of related problems. Unless the complexity of a problem is (considerable) fine-grained, problems of coarser crucial factors can be efficiently encountered in ETHERLISP; two or three times faster parallel execution times can be taken as granted. The possibility of alternative algorithmic schemes provides the development of the proper algorithms for solving the proper problems. We think that the generic algorithms presented (in figures 4-6 and 4-8) can be considered as the best standard vehicle for extracting efficient parallelism in our (or a similar) MIMD system; the experiments revealed that our generic algorithms are clear, compact, highly scalable, and totally transparent. Vital is also the fact that a distributed system should not be rely only on the power provided from the participating processors. An efficient data transportation mechanism is also required. FILOS has been proved as highly efficient, indicating in some extent (table 4.6) that frequent communications is not the dominant disadvantage of distributed algorithms. Finally, we recommend unreservedly the extensive utilization of UDP/IP even when fast execution times are not the primary goal.

Chapter 5

Building Autonomous Abstract Systems on Top of EtherLISP

5.1 Introduction

THE MAIN OBSERVATION OF THE PREVIOUS CHAPTER is that in general real problems perform well when solved in ETHERLISP as the case studies showed. Excluding extreme factors such as the development of ideal algorithms and an overloaded network, we believe that satisfactorily designed algorithms also perform well mainly for two vital reasons. First, the plethora of primitives responsible for executing concrete and short tasks instead of generic ones. Second, the totally transparent and inexpensive in resources FILOS that efficaciously prepares data for a rapid transportation across network links. However, this is not all a concurrent environment should offer. High degrees of flexibility and expressiveness should also be provided. Besides, this is our second dominant aim. We envisaged ETHERLISP as the basis for constructing and experiencing with numerous parallel systems with minimal efforts upon designing and implementing them.

In this chapter several well known, tested, and approved systems are presented and developed. The reader should notice that only a naive implementation is given since we aim to

depict the major aspects of the examined systems on top of ETHERLISP. The reference of each system concludes with a comparison between the particular system and ours to point out the omissions, advantages, and disadvantages of ETHERLISP. Finally, we present the SOCKET package that combined with ETHER results to a system, called ETHERLISP+, providing interprocess communications among user-unrelated processes.

5.2 Objectives

Conjecture 4 gives a concise description of the major aims of this chapter.

■ **Conjecture 4** *We define an autonomous abstract system (AAS) any system built entirely on top of* ETHERLISP *and relying on its own semantics provides a network-based concurrent environment. Our system provides the user with the appropriate tools for an effortless implementation of well defined, tested, and approved AAS's like* RPC *and* LINDA. *Consequently, a deep understanding and consideration of the AAS's fundamental aspects could be achieved as the result of an extended experimentation.*

5.3 Implementing Remote Procedure Calls (RPC)

RPC is a remote procedure call model similar to the local procedure call one. The difference lies in the location of the procedure being called (usually on a distinct machine, or on the same machine but in a different address space). Upon an RPC issue the caller suspends execution, the procedure's parameters (usually locally evaluated) are passed to the server's (callee) environment where the procedure physically resides. At a later time any evaluation result(s) is passed back to the caller which resumes execution. Different RPC approaches may allow other processes executing (including callers) while a caller is suspended, but the caller under service is never allowed to do so (although possible); simply, asynchronous RPC calls violate the semantics of a pure RPC model.

The RPC approach being examined is the one proposed by Birrell [BABN84] and Nelson [Nelson81]. The system has been developed within the Cedar environment where numerous workstations (Dorados) were connected via 10Mbit/sec Ethernets. According to this approach, hereafter called Cedar/RPC, five pieces of program must be invoked to serve a remote call: the *User*, the *User-Stub*, the *RPCRuntime*, the *Server-Stub*, and the *Server*. As

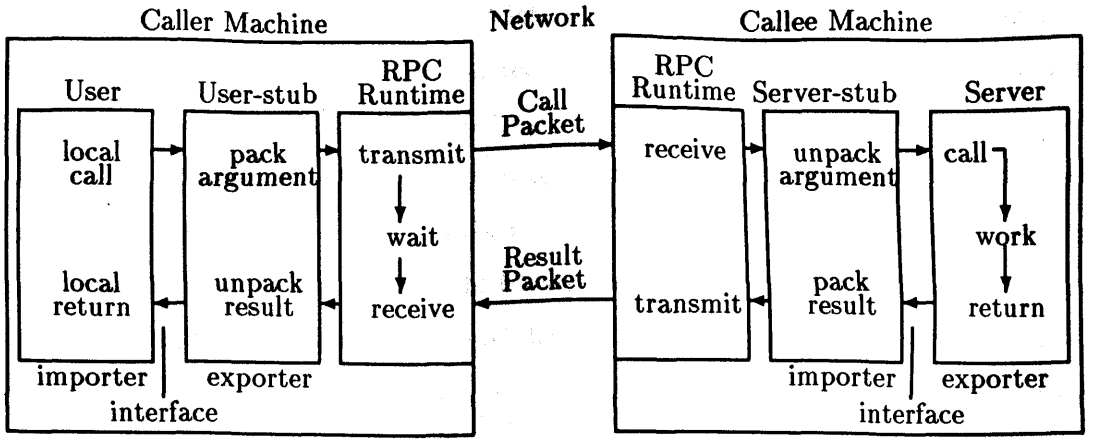


Figure 5-1: The components of Birrell and Nelson's RPC system and their interactions for a simple remote procedure call.

figure 5-1 shows, the User's pieces execute on the caller machine while the Server's ones reside on the callee machine. Both machines exchange information via RPCRuntime instances (the Cedar/RPC's communications package). A remote call requested in the User domain yields the invocation of a corresponding procedure in the User-Stub. The User-Stub encapsulates a specification of the target (remote) procedure and the (local) arguments within one or more packets. RPCRuntime is responsible for transferring them reliably to the target callee. Upon receipt the Server-Stub unpacks the message and invokes the appropriate procedure in Server. Evaluation result(s) travels backwards in a similar manner until it is available to the user. Although complex the mechanism is totally transparent giving the user the illusion of an ordinary local procedure call.

Distributed applications are developed in Cedar/RPC in a fashion assuming the RPCRuntime's support and the code of both User and Server are written as parts of a particular application. The User-Stub and Server-Stub are also application-dependent and generated automatically by the special purpose program *Lupine*. Lupine generates [1] and maintains interface modules as lists describing procedure names along with the types of their arguments and return values. At run-time RPC calls cause the User's code to import an interface whereas the Server's code to export one. A question arises on how a User-Stub is to bind an importer of an interface to an (remote) exporter of an interface. Cedar/RPC's binding scheme implements interfaces as two components: the *type* and the *instance*. The type

[1] At compile-time based on user supplied information.

specifies the abstract (remote) interface, e.g. a mail server, whilst the instance specifies the particular implementor of an abstract interface, e.g. a particular mail server. Issues on locating RPC servers on a network that requires knowledge of IP addresses (section 1.3) will be mentioned in details in section 5.7.

The resembling (communication) semantics of ETHERLISP and RPC together with the information of the Cedar/RPC's structure stated so far, allow the implementation of a simple but complete RPC paradigm (ETHERRPC) as illustrated in figure 5-2. The semantics of LISP permit the automatic construction of the User-Stub, e.g. global variable bindings and function definitions, upon loading an application. Assuming that the appropriate number of servers (remote threads of control (RTC's)) have been created, the function `rpc-lupine()` generates the remote Server-Stubs at run-time. A convenient way to achieve this is when the mapping of functions (RPC services) to Server-Stubs is determined by the user. According to this approach `rpc-lupine()` accepts arguments in the form:

```
((server1 fun1 ... funN) ... (serverN fun1 ... funN))
```

The first element of each group (sublist) refers to the symbolic name of an existing RTC; the rest elements refer to the function names a particular server (RTC) supports; obviously, the bodies of all functions must be transported and defined in the scope of the appropriate servers. The generated interface modules are maintained onto `*server-stubs*`; thus the ith interface module resides on the ith vector's location. Simple RPC calls are issued through `rpc-call()`. As an example consider the following series of statements.

```
1>(progn (setq num1 2 num2 4))
         (defun str-append (s1 s2) (concatenate 'string s1 s2))
         (defun sqr (num) (* num num))
         (defun fact (n) (if (< n 2) 1 (* n (fact (1- n))))) ⟶ FACT

2>(rpc-lupine '((string-server str-append) (math-server sqr fact))) ⟶ NIL

3>(rpc-call '(sqr (* num2 2))) ⟶ 64
4>(rpc-call '(fact (sqr num1))) ⟶ 24
5>(rpc-call '(str-append "abc" "x")) ⟶ "abcx"
6>(rpc-call '(+ 1 2 3)) ⟶ 6
7>(rpc-call '(sqr (rpc-call '(sqr num1)))) ⟶ 16
8>(rpc-call '(* (sqr num1) (fact num2))) ⟶ 96
```

The first statement defines the (local) User-Stub and the second one two remote Server-Stubs named as `string-server` and `math-server`. The parameters of statements 3, 4, and 5 are evaluated locally and then an acquisition for a remote service is applied. Recall that the network path is automatically generated through `*server-stubs*`. In ETHERRPC only

the arguments of a call must be packed in terms of evaluating the arguments in their original (local) lexical scope; upon receipt, calls are ordinary function calls and hence applicable in an *"as-it-is"* format to any remote Lisp evaluator (RTC). More precisely, unpacking arguments and packing result(s) remotely, and unpacking result(s) locally for any single call, as figure 5-1 shows, is not performed. The astute reader may have noticed that the domain in which user-defined functions are defined is passed as an argument to `rpc-lupine()`. Thus any user function call is applied to the proper Server-Stub; otherwise an error signals a non-available service. As an RTC is a complete Lisp evaluator all built-in functions can be applied as RPC calls (statement 6); in such a case a Server-Stub is arbitrarily selected. Finally, statement 7 denotes the support of nested calls that occur in a serial order from left to right. A more sophisticated version of `eval-rpc-call-arg-lst()` would allow the evaluation of RPC calls' arguments as independent RPC calls (statement 8); note that arguments (RPC calls) referring to different Server-Stubs is also easy to implement.
However, the serial execution of remote calls (as transactions) entails to a transparent distributed programming which is however inefficient owing the lack of asynchrony. The RPC model [SunOS] is (very) useful only for constructing distributed operating systems with capabilities of dispersing easily accessible services (daemons) around computer networks.

```
(defun rpc-call (rpcall)
  (let ((call-name (car rpcall))
        (target-server-stub-found-flag)
        (target-server-stub))
    (when (not (functionp call-name))
          (error "~S is not a function." call-name))
    ; If a user-defined service...
    (if (equal (symbol-package call-name) (find-package "USER"))
        (progn
          ; Generate call's path (find call's User-Stub).
          (dotimes (i (length *server-stubs*))
            (let ((target-server-stub-record (aref *server-stubs* i)))
              (when (find call-name target-server-stub-record)
                    (setq target-server-stub (elt *ether-list* i))
                    (setq target-server-stub-found-flag t))))
          (when (not target-server-stub-found-flag)
                (error "The function ~S is system-wide undefined." call-name)))
        ; For built-in calls select an RTC non-deterministically.
        (setq target-server-stub (elt *ether-list* (random (length *ether-list*)))))
    ; Evaluate call's arguments.
    (let* ((call-args (eval-rpc-call-args (cdr rpcall)))
           (request-call (cons call-name call-args)))
      (push-ether request-call target-server-stub)
      (listen-ether target-server-stub))))

(defvar *server-stubs* (make-array (length *ether-list*)))
```

TO BE CONTINUED

```
(defun rpc-lupine (&rest args)
  (dolist (elt args)
    (let* ((target-server-stub (eval (car elt))) ; Target RTC.
           (target-server-stub-id (ether-id target-server-stub))
           (fun-lst (cdr elt)) ; Services.
           (ack-fun-lst))
      (when (not (etherp target-server-stub))
            (error "~S is not a server." target-server-stub))
      (dolist (fun fun-lst)
        (when (not (functionp fun))
              (error "~S is not a function." fun))
        ; Get function's body and prepare it for transmission.
        (let ((target-server-stub-entry (cons 'DEFUN (cdr (symbol-function fun)))))
            (push-ether target-server-stub-entry target-server-stub)
            (push (listen-ether target-server-stub) ack-fun-lst)))
      (setf (aref *server-stubs* target-server-stub-id) ack-fun-lst))))

(defun eval-rpc-call-args (rpccall-args)
  (let (rpccall-arg-lst)
    (dolist (arg rpccall-args) (push (eval arg) rpccall-arg-lst))
    (reverse rpccall-arg-lst)))
```

Figure 5-2: Code of an abstract RPC interface on top of ETHERLISP.

5.4 CMU Common Lisp

Lott and Chiles in [MacLach92] present the recent implementation of CMU Common Lisp (CMU/CL) developed in the Computer Science Department of Carnegie Mellon University. The system is currently supported on several machine architectures including, MIPS-processor DECstations, SPARC-based SUN workstations, and the IBM PC RT. The backbone of CMU/CL is the standard COMMON LISP as is defined in [Steele90]. Various extension packages, such as a source level debugger, an interface to UNIX system calls, and a foreign function call interface, have been embedded. Our interest focuses on comparing our ETHER environment (section 2.3) against the Interprocess Communication (IPC) and RPC extensions of CMU/CL.

The networking capabilities of CMU/CL are based on two packages. The *remote* package provides an RPC facility including interfaces for creating (remote) servers, connecting to already existing ones, and calling functions in other Lisp processes. The *wire* package provides control for sending data structures along *wires* (abstract entities used for underlying I/O socket operations (section 1.5)). The *remote* package resides on top of this package. Creating and connecting remote servers follows the trivial pattern although the entire procedure is not highly transparent. The client must retrieve the server's IP address through the operating system's host database (using multiple function calls), whilst the specific server's

port number must be known to the client. Both information (IP address and port) are transmitted along with the connection request. A process upon turning to a server picks up a random 16-bit unsigned integer to use it as the receiving port. If the port is not available for any reason an error is signalled causing the interruption of the procedure. Consequently, the finding of a free port must be an iterative process, and as the implementors mention, the user is required to write a loop catching conditions of type error (probably to avoid program's interruption). We confronted this problem in a completely efficient and transparent fashion. The primitive responsible for finding free ports returns either a 16-bit integer or `nil` denoting that either a port has been found (and returned) or another try is needed. After one or several tries a port is located and reserved. We use this technique in our SOCKET package described in section 5.7; in addition, the reserved port is globally accessible through the Socket Address Service (SAS). After a connection has been established numerous RPC requests are allowed. Again, transparency is missing; the client must explicitly signal the server to make the evaluation result(s) available at the client's site. Each transmission at RPC level requires the flushing of all internal buffers associated. If a remote process leaves any input (unread) on a *wire* object, when data has been transported via the *wire* package, the data of a subsequent RPC request will be mistaken causing unknown lossage. Apparently, the absence of harmony between the IPC (*wire*) and RPC (*remote*) packages prevents the user to take advantage from queuing requests; hence, asynchrony in CMU/CL is prevented.

Only a very limited number of Lisp objects can be sent as arguments to remote function calls or as return values. In fact, only 32-bits integers, symbols, and lists can be transported. Sending other objects implies their representation in terms of the above three basic types. For example, to send a string over a wire `prin1-to-string()` is used locally, the string is transmitted, and the explicit use of `read-from-string()` is remotely required. Though unclear, the transmission of a string is obscure and entails substantial overhead. A symbol is transmitted as two strings, its package name and its print name. This compares with FILOS, which efficiently encodes (in 2 bits) symbols' packages within symbol blocks (figure 3-2), and hence the receiver avoids looking up internal tables when interning first-time-received symbols. The existence of *remote-objects* in CMU/CL hints the incomplete support in transporting many data types. Any Lisp object can be converted into a remote-object by means of representing data structures as unique tokens (integers). What is actually transferred is the tokens. By this technique remote processes refer indirectly to

local structures without allowing any operation on them. Finally, both ends of a communication channel (wire) must exactly know the types of incoming and outcoming data, as well as their direction. Thus, transported data are explicitly written/read by the appropriate complementary (type-dependent) encoding/decoding routines at local/distant locations. Obviously, CMU/CL does provide interprocess communication but both efficiency and transparency are very poor. We believe that the choice of the Sun's RPC and XDR facilities (see section 5.7.5) as built-in tools causes CMU/CL to inherit many of its disadvantages.

5.5 Avalon/Common Lisp

Avalon/Common Lisp [Clamen89] is another intergrated network-based system much like ETHERLISP. Computations are performed over a distributed set of evaluators each executing a LISP image at a distinct node. A node may be host for multiple evaluators but only one thread of control executes within an evaluator at once. A computation starts from an initiating evaluator that keeps track of the paths (channels) connecting other remote evaluators. Communication is achieved in terms of remote procedure calls with call-by-value semantics. Avalon/Common Lisp (A/CL) provides support for persistent transactions; remote evaluators (servers) maintain private recoverable stores that encapsulate all bindings explicitly declared as persistent (ordinary bindings are volatile). Thus, persistent bindings survive host crashes whereas volatile ones do not.

```
(defvar *remote-evaluator* (determine-remote-evaluator)) ; Select remote server.

(defmacro remote (exprs &optional (rem-evaluator *remote-evaluator*))
  (let ((local-count (gensym)))
    `(let ((,local-count (parse exprs :locals t))) ; Count potential locals.
       (push-ether (parse exprs) ,rem-evaluator)   ; Scan request.
       (dotimes (i ,local-count)                   ; Serve call-backs (if any).
         (push-ether (parse (listen-ether ,rem-evaluator)) ,rem-evaluator))
       (listen-ether ,rem-evaluator))))            ; Get final result.

(defmacro local (expr)
  (if *remote*
      (progn (push-ether expr) ; Make a call-back.
             (listen-ether))   ; Return local's binding.
      (error "Local execution of macro LOCAL is not allowed.")))
```

Figure 5-3: The basic constructs of Avalon/Common Lisp on top of ETHERLISP.

The most interesting aspect that also characterizes A/CL relates to the manner in handling distributed computations. Two special variables allow the switch of the computation

thread from the designated local evaluator to some remote one. In particular, the remote evaluator handles expressions of the form (**remote** *expr*) whilst the local one expressions of the form (`local` *expr*). The semantics of `remote()` specify: (a) The actual computation is performed either by the (remote) evaluator bound to `*remote-evaluator*`, or by the one passed as an optional argument. The computation request is transferred along with its lexical environment. (b) The object returned is a copy of the result as opposed to the result object itself. The special form `local()` is meaningful only within the scope of a `remote()` statement; it is also dynamically evaluated only in the scope of the evaluator bound to `*local-evaluator*` (the requester) after the remote evaluator has received the request. The later is called a *call-back*. Figure 5-3 demonstrates an implementation of the basic constructs of A/CL. Although both primitives should be very similar in structure with the original ones, our implementation of the function `parse()` is considered naive. However, our goal is to show that ETHERLISP's IPC architecture permits a rather effortless coding of both A/CL's novel aspects. The underlying philosophy of A/CL (hidden in `parse()`) is a matter of trivial coding. The macro `remote()` first invokes `parse()` for counting any potential `local()` calls in a request. Before transmission the request is once more scanned by `parse()` for encapsulating the request's lexical environment. Then enters in a loop for serving (if any) pre-counted call-backs; that is, whenever a `local()` is remotely encountered an automatic (via network) request replaces `local()` with its binding retrived from the sender's scope and the computation resumes remotely. For example, the statement

`(let ((a 123) (b 45)) (remote (+ (local a) b))) `$\xrightarrow{a}$` (+ (local a) 45) `$\xrightarrow{b}$` (+ 123 45)`

transmits the expression pointed to by the arrow a, while arrow b indicates what the remote evaluator finally computes after a has been replaced by its value 123. The final result is returned by the last network-read operation. Our implementation of A/CL disclosed a few inefficiencies. Frequent and inconsiderate use of *locals* within remote requests along with the sequential nature of transactions inevitably degrades the performance, especially when the granularity of access is very fine. Functions other than the built-in ones require transmission of their bodies as part of their local lexical environment. Consider the form:

```
(labels ((foo (x) ... (foo (1- x)) ... ))
        (dotimes (i large-num) (remote (foo (local i)))))
```

The `foo()`'s body is transferred `large-num` times where the same number of call-backs must be performed. It should be noticed that `foo()` is a recursive function that perplexes and burdens communications. The problem is more apparent because objects are trivially transmissible. The sender simply pass the objects' print representation (as produced by

the Lisp writer) and the receiver reconstitutes a copy using the ordinary Lisp reader. This approach is similar to our SIO mechanism that in section 3.10 has been proved inadequate. Finally, asynchrony is very limited. Evaluators are bound in pairs and each pair awaits transactions to commit implying that only one pair is active at once. Hence, simultaneous evaluation of multiple requests is prevented. Meanwhile, successive computations directed to distinct remote servers require the same in number bindings of explicitly selected pairs at run-time.

5.6 Implementing LINDA

Linda [NCDG89, NCDG86] is another paradigm for parallel programming. Its primary concepts rely upon the simplicity, portability, and scalability derived from a small set of operations. These operations when embedded in a language yield a new parallel language. The added semantics cause no alterations or conflicts with the original (and remaining) ones. As an evidence, Linda has been embedded in several popular LISP dialects as the work in [UDNMcD90] and [White88] evinces. Our aim in this section is only to describe the fundamental primitives of Linda, since the model is to be extensively examined and compared with our model PRAXIS in the next chapter.

Linda's theory introduces the notion of the *tuple space (TS)* that is a shared store accessible by multiple co-processes simultaneously. The TS contains any number of tuples which are either *passive* or *active*. Passive tuples encapsulate numerous value fields - in some implementations [Leich89] the number is limited - specifying either raw data (actuals) called tuples, or patterns (formals) called templates. Interprocess communication is achieved by *pattern-matching*, that is matching formals in templates with the actuals in tuples. Two tuples match if they have the same number of arguments (arity) and each pair of corresponding fields match. Two fields match if they are of the same data types, and either both are actuals with the same value, or one is an actual and the other is a formal. Clearly, the TS is an associative memory where objects are accessed by pattern-matching instead of their associative addresses. Active tuples instantiate processes that after execution turn into passive tuples and inserted into TS. A passive tuple t is inserted into TS by the *out(t)* operation. All tuple's fields are evaluated prior to insertion. The *in(m)* operation returns any tuple t that matches the template m. *in()* blocks until a matching tuple is found in which case it is removed from TS. If there are more than one matching tuples one is selected non-deterministically. Tuples are also extracted from TS through the *rd(m)* operation but

the matching tuple remains in TS. *eval(t)* is similar to *out()* but the tuple *t* is evaluated after it has been inserted into TS. The evaluation of such a tuple imposes the creation of a new process.

```
(defun make-tuple (&rest tuple-args)
  (when (not (typep (car tuple-args) 'string)) ; Tuple names are strings.
        (error "Invalid tuple tag ~S." (car tuple-args)))
  tuple-args) ; Return tuple as a list.

(defun linda-out (tuple)
  (if *remote* ; On an RTC enclose an OUT indicator for scheduling purposes.
      (let ((out-tuple (cons *out-operation* tuple)))
        (push-ether out-tuple)) ; Resume execution.
      (insert-tuple-in-pool tuple))) ; On the LTC insert the tuple into TS.

(defun linda-in (pattern)
  (if *remote* ; ON an RTC enclose an IN indicator for scheduling purposes.
      (let ((in-tuple (cons *in-operation* pattern))
            (matching-tuple))
        (push-ether in-tuple)
        (setq matching-tuple (listen-ether)) ; Wait for a (locally) matching tuple.
        (replace-formals-in-pattern pattern matching-tuple))
      (loop (let ((matching-tuple (match-tuple-in-pool pattern)))
              (when matching-tuple (return-from linda-in matching-tuple))))))

(defun linda-eval (eval-fun-name &rest eval-fun-args)
  (prevent-remote-execution) (check-function eval-fun-name)
  (let ((scheduled-ether (car *ether-list*)) ; Get the next available RTC.
        (active-eval-tuple (cons eval-fun-name eval-fun-args)))
    (push-ether active-eval-tuple scheduled-ether)
    (rotate-ether-list scheduled-ether)))    ; Find the next available RTC.
```

Figure 5-4: Code for the basic LINDA primitives when operating on a single global TS.

The implementation of Linda on top of ETHERLISP, called ETHERLINDA, according to the structure proposed by the original implementors Carriero and Gelernter was a rather simple procedure. In figure 5-4 the code implementing the basic Linda primitives when operating on a single global TS is only given. The code related with tuple matching and process scheduling is bulky and its full demonstration is beyond the scope of this chapter. According to our (first) implementation tuples may have an arbitrary number of fileds as **make-tuple()** states. Note that the first field of each tuple is a literal string which names tuples. Tuples with the same names belong to conceptually separated tuple groups. When **linda-out()** performs on an RTC the *outed* tuple is inserted in TS, maintained from the initiating LTC, via network. Local execution is also allowed because in some applications (remote) processes are blocked awaiting some initial data. In the same way **linda-in()** is constructed. At a remote location a process blocks until a matching tuple (found locally)

is received in which case the formals are replaced with actuals. Local execution is allowed when the final results are to be extracted and be available to the user; besides, any remaining tuples (handled by *rd*'s for example) should be removed when an application terminates. We handle active tuples slightly different. We consider active tuples any Lisp function along with an arbitrary number of arguments. When `linda-eval()` is applied a new process is created by means of instantiating a free RTC which executes the function specified in the tuple. Since ETHERLISP allows one process per processor Linda processes are dispatched according to the predetermined number of already existing RTC's in a Round-Robin scheme. Obviously, the number of RTC's is application-dependent. The structure of our system also implies only one processor that initiates and controls multiple RTC's. Therefore, Linda processes can be scheduled only form the single local LTC (root). That is why remote execution of `linda-eval()` is prevented remotely.

```
(defun linda-server ()
  (linda-in (make-tuple "work-items" '?value-1 '?value-2))
  (linda-out (make-tuple "add-result" (+ value-1 value-2)))
  (linda-server))

(defun linda-client (list1 list2)
  (let (sum-list)
    (dotimes (i (length list1))
      (linda-out (make-tuple "work-items" (elt list1 i) (elt list2 i)))
      (linda-in (make-tuple "add-result" '?sum-value))
      (push sum-value sum-list))
    (linda-out (make-tuple "sum-list" (reverse sum-list)))))

(defun main (data-list1 data-list2)
  (linda-eval 'linda-server)                          ; Instantiate SERVER on RTC 0
  (linda-eval 'linda-client data-list1 data-list2)    ; Instantiate CLIENT on RTC 1
  (linda-schedule "sum-list" '?sum-list)              ; Start scheduling on LTC
  (format t "~%Car-wise sum: ~S" sum-list))           ; Print the final result
```

Figure 5-5: An example LINDA program developed in ETHERLINDA.

Figure 5-5 demonstrates a simple program developed in ETHERLINDA. **`main()`** creates and dispatches two processes, `linda-server()` and `linda-client()`, which execute in parallel. **`linda-server()`** continuously accepts two integers (the *car*'s of two data lists) and returns their sum. Data are obtained from `linda-client()` which inherits it upon its creation. Sums are collected from `linda-client()` and placed in a list which is inserted into TS as the final program's result. Communication is guaranteed by **`linda-scheduler()`** that shares TS between the two processes. After the server completes an *in()* the patterns **`?value-1`** and **`?value-2`** are replaced by two actual numbers of type integer stored in the

two (automatically) generated variables **value-1** and **value-2**. It should be noticed that both processes remove tuples when *in*ing them. Hence, the TS will contain only one tuple (the list of sums) when the program terminates normally. Since many tuples may be in TS after a program completes normally, the scheduler has been designed in such a way so that it accepts as parameters the name and template of the result tuple like an ordinary *in()*. Any remaining tuples are considered fossils and removed leaving to subsequent applications a "clean" environment. However, the program is not optimal since TS would be filled up by **main()** locally without using the network, and letting **linda-server()** as a single (remote) process picking up tuples as in figure 5-5. The structure of the program as presented aids to a better demonstration of the Linda's semantics, as well as it constitutes the appropriate environment for reasons being mentioned in the next section.

5.6.1 On Applying FILOS on LINDA Tuples

We believe that Linda is the most representative paradigm reflecting the necessity of a message compression mechanism. Suppose that both data lists of the program in figure 5-5 are of length 1000 elements, and each element is a 5-digit integer. Consequently, the four tuples are transported 1000 times each. Table 5.1 shows the byte size of each tuple, as well as its total byte size when SIO (section 3.10) and FILOS are in effect. In particular, the size of the first tuple is 32 bytes yielding a total 32000 bytes SIO size; FILOS results to a 34-bytes message, when the first tuple is transmitted for the first time, and a 5-bytes message for the rest 999 transmissions; totally 5019 bytes. Clearly, the 84.31% of the SIO byte size is excluded. FILOS results to a "poor" 54.96% compression performance in case of the second and third tuples because the actual fields are each time different (and literally) encoded integers. Clearly, the total communication byte-overhead has been reduced to 70.83%.

Tuple Case	*Size*	SIO	FILOS
1: ("work-items" ?value-1 ?value-2)	32	32000	5019 (84.31%)
2: ("add-result" <5-digit>)	20	20000	8011 (59.94%)
3: ("work-items" <5-digit> <5-digit>)	26	26000	13000 (50.00%)
4: ("add-result" ?sum-value)	25	25000	4009 (83.96%)
Total tuple size (×1000) in bytes	103	103000	30039 (70.83%)

Table 5.1: Compression performance on LINDA tuples.

In completion of the above, one should note the benefits of the object creation and interments which is approximately 99% less than the required. Moreover, the byte-overhead

would be much further reduced when tuple fields are objects that may be replaced by SYS_i and HSH_i indices. These observations reveal that FILOS encourages the use of long tuple names (tags) and field names that contribute to a self-identifying code essentially for large applications. The frequent transmission of pattern tuples is also encouraged since their fields are just place-holders almost surely to remain encoded ATM_i indices for the very future. The later is true due to the general LINDA property that any two complementary tuples are labeled with a common tag. In our example, the tags `"work-items"` and `"add-result"` are common for the tuple pairs [1,3] and [2,4]. Consequently, the tuple space could be distributed among processes since pattern-matching is not expensive. The congestion caused by a single global tuple space manager would be avoided [UDNMcD90, p:12] and the entire system would be less dependent.

5.7 The EtherLISP's SOCKET Package

The problem of locating remote servers' IP addresses was left unanswered in the previous RPC's survey. It is obvious that the architecture of ETHERLISP is only capable for providing process-to-process communication within the single domains of unrelated users. Communications among processes in different user domains (user-to-user) gives rise to the need for a system that allows the direct use of underlying networking facilities on user's behalf. Such a system (hereafter called Sun/RPC) is the one presented in [IRIS-4D87] as a tool for professional usage. Providing low-level networking mechanisms (such as TCP/IP, UDP/IP and sockets) Sun/RPC is considered as the high-level communications paradigm for distributed operating systems (such as IRIX 4.3BSD and SUNOS 4.2BSD). In distributed operating system RPC terminology, a *server* is a machine, a *service* is a collection of remote program(s), and a *remote program* encapsulates one or more remote procedures. The procedures and the types of their arguments and results are documented in the specific program's protocol specification. Finally, a *client* is the program initiating RPC calls to services. Sun/RPC's structure consists of three general parts: (i) The ***highest layer*** which is totally transparent to the operating system, network, and machine on which it is related. At this level an RPC call is actually the invocation of an (permanently registered) RPC program which executes on some server and returns some result(s). (ii) The ***middle layer*** that is still highly transparent by means of using a few RPC constructs for making RPC calls. Operations of the fundamental constructs include, the obtainment of a system-wide unique procedure identification number (program's registration), and the actual execution

of RPC calls. (iii) The *lowest layer* that is suitable for serious RPC programming. Applications of this layer can manipulate sockets (e.g. creation) and alter the defaults of the RPC constructs (e.g. TCP/IP instead of the default UDP/IP). Recalling that UDP/IP is the default communications protocol, Sun/RPC acts as an ordinary RPC paradigm. That is, a client invokes a procedure to send a data packet (UPD-datagram) to a server. Upon receipt the server invokes a dispatch routine, services the call, sends back the result(s), and the control returns to the client. Note that clients are dormant while awaiting to be served. The SOCKET package [2] has a dual usage in ETHERLISP. It can be used either as an independent environment or together with ETHER in which case turns to a more general and powerful system called ETHERLISP+. The SOCKET package as a stand-alone environment provides the basis for developing any sort of network-based application. For instance, ETHERLISP was initially implemented relying on the SOCKET's primitives. However, the final version is totally written in C to evade the usless burden owing to Lisp's (very expensive) READ-EVAL-WRITE filtering of each primitive. The SOCKET package consists of three modules (services) in terms of providing support for TCP/IP, UDP/IP, and SAS (a service responsible for manipulating socket addresses).

5.7.1 Reliable Communication Service (RCS)

The complete TCP/IP transport protocol is provided by the RCS module. Figure 5-6 demonstrates the basic procedure of the fundamental RCS primitives. Note that the number prior to the Lisp's prompt specifies the order primitives should be applied. A reliable connection (virtual circuit) utilizes three sockets. Two sockets implement the endpoints of a channel providing the third one which creates the receiving endpoint. Upon creation of a (first class) *listener* object its IP address [3] is needed for directing connection requests to it. Statement 3 blocks until a connection request is captured in which case a first class ***passive socket*** object is returned. Connection requests can be made from a different address space (on the same or different machine) providing the target IP address (statement 4). After a connection has been successfully completed a first class *active socket* object is returned and used for sending data. When a connect session terminates all sockets must be closed (killed). This is important since occupied port numbers are freed for future re-use. In addition, the

[2] This package came as the result of the first research's stages along with the implementation of SIO. Moreover, a similar package has been developed providing the C language with interprocess capabilities. The system, called ETHERC, is presented along with some demonstrative examples in Appendix C.

[3] The system-wide unique machine name and port number are enough for distinguishing passive sockets; the term passive refers to sockets used only for receiving data.

number of simultaneously open TCP/IP connections a user can have is limited by the operating system. Attention is needed when sockets are closed. For example, an active socket must be closed before the corresponding passive one; or, having the endpoints closed (freed) the alive listener can create a new channel by using the same IP address. This is essential for avoiding the overhead for creating and seeking for server IP addresses.

5.7.2 Connectionless Communication Service (CCS)

The CCS module utilizes the unreliable but flexible and rapid UPD/IP transport protocol. The use of the basic CCS constructs is abridged in figure 5-7. The simplicity of UDP/IP is reflected from the simple manner CCS connections are established, e.g. the absence of a listener. The most important aspect of CCS is that a first class *catcher* object accepting both connections and data from multiple *thrower* objects residing in irrelevant contexts.

5.7.3 Socket Address Service (SAS)

A common requirement of the RCS and CCS modules is the knowledge of the IP addresses of passive sockets. SAS is a service dedicated for supplying socket addresses on request in a totally transparent and efficient fashion. Requisite presupposition of services like SAS is a permanent IP address. This can be achieved by reserving a specific port number on a specific machine, since the first $2^{10} - 1$ (0 to 1023) ports are reserved for either existing operating system services, or for services being developed. A permanent address is consequently a well-known address to which connections from any network site can be made.
Figure 5-8 shows the basic procedure for using SAS when CCS is in effect. Upon creation, a passive socket is assigned a free port selected within the range [2^{10} to $2^{16} - 1$]. Note that the port 2363 of a TCP/IP socket is different from the port 2363 of a UDP/IP socket both created on the same machine. Statement 2 shows the way socket addresses are globally advertised. A server program sends a *propagation* SAS request specifying a receiving IP address and a system-wide unique socket name, used (by SAS) as a key for selecting among its entries. Addresses are available (statement 4) to every network location by a *get* SAS request which specifies the name (X in the example) of the seeking socket. The fact that SAS's default protocol is UDP/IP, SAS operations accomplished by CCS primitives are less complex and complete faster. The reason is that `propagate-address()` creates a socket Y, connects it to the SAS's address, propagates X's address via Y, and then uses the (already created) socket X for receiving confirmation for the operation's success.

(Local) Host: "cubby"	(Remote) Host: "balin"
1>(setq l (make-listener)) #<LISTENER 3,1> 2>(setq lid (listener-id l)) ("cubby" . 1896) 3>(setq s (socket-listen lid)) #<SOCKET 4,3> 7>(setq msg (socket-read s)) (A B C) 8>(progn (close-socket s) (close-listener l)) NIL	4>(setq s (socket-connect '("cubby" . 1896))) #<SOCKET 3,3> 5>(setq msg (socket-write s '(a b c))) (A B C) 6>(close-socket s) NIL

Figure 5-6: Reliable Communication Service (RCS): The Basic Procedure.

(Local) Host: "cubby"	(Remote) Host: "balin"
1>(setq ctr (make-catcher)) #<CATCHER 3,1> 2>(setq msg (catch-message ctr)) 3.1415926535897931 >(close-catcher ctr) NIL	3>(setq thr (connect-thrower '("cubby" . 1588))) #<THROWER 3,1> 4>(throw-message thr pi) 3.1415926535897931 >(close-thrower thr) NIL

Figure 5-7: Connectionless Communication Service (CCS): The Basic Procedure.

(Local) Host: "watt"	(Remote) Host: "spot"
1>(setq x (make-catcher)) #<CATCHER 3,1> 2>(propagate-address (catcher-id x) 'x x) ("watt" . 3687) 3>(catch-message x) #C(2/3 3) >(close-catcher x) NIL	4>(setq addr (get-address 'x)) ("watt" . 3687) 5>(setq y (connect-thrower addr)) #<THROWER 3,1> 6>(throw-message y (complex 2/3 3)) #C(2/3 3) >(close-thrower y) NIL

Figure 5-8: Socket Address Service (SAS): The Basic Procedure

In case of RCS SAS requests the receiving socket X had to be created (and closed). Other important services provided by SAS include seeking for all UDP/IP or TCP/IP addresses either on a specific host or system-widely. SAS as any data-base system provides operations such as inserting, deleting, and renaming entries. Our SAS service is similar to the Sun/RPC's Port Mapper Program Protocol (PMPP) which maps RPC program and version numbers to transport-protocol-specific port numbers.

5.7.4 EtherLISP+ and Sun/RPC

ETHERLISP+ permits the development of applications configured as illustrated in figure 5-9. Users are allowed to create any number of private RTC's. Communication among different users is allowed via global servers whose IP addresses are extracted by querying SAS. In all cases requests directed either to SAS or to generic servers are of the form (in Lisp notation) `(reply-address . request)`, obviously for correlating replies to requests. For instance, the following code would implement a temporary CCS service request from some generic server. The function can be initiated from any LTC or RTC of some user providing the unique name (and not the IP address) of the generic server being accessed. Note that `on-connect` specifies any valid Lisp expression.

```
(defun ccs-connect-to-generic-server (gen-srv-name &key on-connect)
 (let* ((input-socket (make-catcher))
        (input-socket-id (catcher-id input-socket))
        (gen-srv-address (get-address gen-srv-name)))
   (let ((output-socket (connect-thrower gen-srv-address))
         (message (cons input-socket-id (list on-connect))))
     (throw-message output-socket message))
     (let ((gen-srv-reply (catch-message input-socket)))
       (close-catcher input-socket) (close-thrower output-socket) gen-srv-reply)))
```

Sun/RPC and ETHERLISP+ are *generic systems* that "glue" user-unrelated processes running in different address spaces. On the other hand, single-user-dedicated systems like ETHERLISP provide higher degrees of transparency and simplicity without loss of efficiency and expressiveness. We advocate that the provision of private RTC's in generic applications (figure 5-9) yields an efficient subenvironment in which user applications perform well. Usage of generic servers can be optional and whenever interuser communications are desirable. A comparison between Sun/RPC and ETHERLISP bore out the following imperfections owing to the philosophy of generic systems.

Sun/RPC utilizes UDP/IP as the default transport protocol. As an effect, RPC calls are restricted to $8K$ bytes of data bulk. The user must intervene in the lower layers of Sun/RPC for setting TCP/IP as the default transport protocol. Thus, larger data volumes in terms of "arbitrary" byte sequences could be transported. ETHERLISP (and ETHERLISP+) utilizes both protocols transparently but in any case our underlying segmentation mechanism

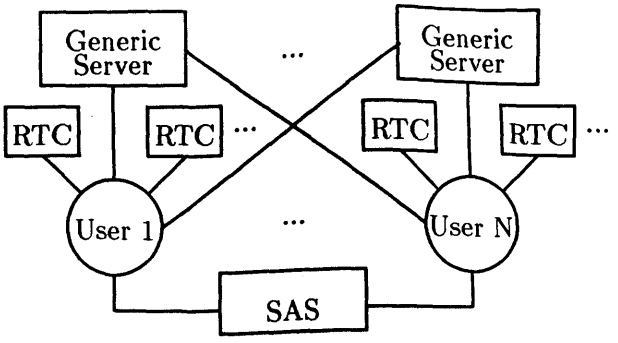


Figure 5-9: The structure of a generic distributed application in ETHERLISP+.

permits data structures of arbitrary length and complexity to be sent across network links. A network-wide unique identification number (id) is enclosed in every Sun/RPC request for matching replies to requests. ETHERLISP avoids this burden (in terms of transmission cost and id management) since each server (RTC) is distinguished by its system-wide unique communication channel id or name.

5.7.5 XDR and FILOS

Another important aspect of generic distributed systems is the manner data are prepared for transportation across network. To this survey we collate FILOS and XDR [SunOS, Vol:10]. The eXternal Data Representation (XDR) standard is the backbone of the Sun/RPC package. A set of XDR routines allow programmers to describe and transport arbitrary data structures across different languages, operating systems, and machine architectures. XDR standarizes data representations in a *canonical* fashion, e.g. a single byte order (big-endian) and a single floating-point representation (IEEE standard). The advantage of the canonical approach is its simplicity derived from a strictly (and once) defined underlying conversion routines. The absence of control over these routines in some cases results to redundant data conversions. For instance, a little-endian sender machine converts integers to the XDR's big-endian byte order, while a little-endian receiver machine performs the opposite.
XDR requires explicit description of the data structures being transported (no explicit data-typing). For example, the sender employs the specific encoding routine that converts objects of type string into the proper transmittable format; the receiver must know that such an object is to be expected and the decoding routine which handles strings is applied. Conversely, FILOS recognizes data structures and based on their attributes automatically

encodes and decodes them. It is clear that ETHERLISP and Sun/RPC, as any physically distributed system, requires a strict synchronization between corresponding sending and receiving points. Sun/RPC additionally requires a strict object type synchronization fact that substantially restricts the development of abstract applications.

XDR always converts quantities to 4-bytes multiples when encoding (serializing). That is, if data is encapsulated within n bytes, when n is not a multiple of four, then $r \in [1,2,3]$ residual zero bytes are appended. Thus, unless explicitly packed (compressed), quantities of type character for example would occupy 32 bits each. Oppositely, FILOS provide a more cost-effectiveness packing scheme since the smallest block is of size 1-byte. However, FILOS will soon support an improved version of the current packing scheme. Elementary quantities of data structures, such as characters of either print names or strings, will occupy the necessary number of bits (section 7.3).

Apart from serializing and deserializing data structures, XDR is also responsible for allocating and freeing memory explicitly. FILOS does allocate memory in a transparent way but freeing is automatically (and transparently) performed by the system's garbage collector.

5.8 Summary

Apart from an efficient concurrent environment ETHERLISP also provides the basis for an effortless developing of well-known parallel paradigms. This is vital since users can get the advantage of experimenting and experiencing with numerous paradigms in short time periods. It should be noticed that other paradigms such as the Parallel Virtual Machine (PVM) [Sunder90] and the TIME WARP [Jef85] model can be also implemented easily. However, it is essential that ETHERLISP and ETHERLISP+ are well defined systems surpassing in many cases of other accepted systems. Meanwhile, the SOCKET package converts ETHERLISP into a generic system providing user-to-user communications.

The experience gained from the examination of numerous distributed systems including ours along with an efficacious message transportation mechanism, led us to develop an alternative paradigm called PRAXIS as we will see in the next chapter.

Chapter 6

Implementing Innovative Techniques on Top of EtherLISP

6.1 Introduction

COLLECTIVELY WE HAVE SHOWN in the previous chapters that ETHERLISP is a network-based system which provides the basic tools for the development of compact, scalable, and efficient parallel algorithms. Apart from its novel method of encoding/decoding and compressing messages our system more or less provides a simplistic distributed environment. Conversely, concurrent paradigms such as LINDA [NCDG86], FUTURES [Halst85], and TIME WARP [Jef85] introduce their own particular semantics that characterize them and make them differ from others. In this chapter we introduce some techniques that provide ETHERLISP with innovative semantics. The first technique allows explicit clustered interprocess communications which, under some circumstances, contributes to a more efficient scheduling and faster running time of multiple co-processes. The second technique can be considered a complete paradigm on its own right, called PRAXIS; its clear and expressive semantics along with a simple underlying scheduling mechanism yield compact, scalable, particularly flavoured, and in some cases prominent solutions to several problems.

6.2 Batch Message Passing (BMP)

The first novel feature of ETHERLISP is the injection of the *Batch Message Passing*, BMP for short, technique that provides explicit clustered message communications. BMP can be thought of as a collection of logically related messages that correspond to the evaluation results originated at discrete time units from multiple remote threads of control (RTC's) and transmitted as clusters. The primary reason that triggered the development of BMP is the impossibility of inspecting the precise number of pending messages at the receiving endpoints of communication channels with the standard Ethernet. Indeed, apart from only inspecting pending messages and read them in a FIFO order both UDP/IP and TCP/IP protocols do not permit any other operation, at the user's level, on the underlying message queues. Instead, the internal structure of these protocols, as presented in [SunOS, Vol:10], should be properly enhanced to meet the needs of individual communication protocols that is far in excess to the purpose of this research. However, we advocate that in many cases the precise knowledge of the number of pending messages before they are read (processed), would lead to a more efficacious scheduling and work-load balancing policies of multiple concurrently executing components of a distributed system; meanwhile the overall performance of the system would speed up substantially.

The basic concept of BMP requires the enclosure of critical communication regions (of RTC's only), e.g. a code segment including either a *push*, or a *listen*, or both message operations, between `ether-lock-output()` and `ether-unlock-output()`. The first primitive instructs the remote sending primitive to redirect its output to a queue (*batch* mode) instead of its standard output channel (*standard* mode); the second one inserts the number of queued messages at the head of the queue and transmits the entire queue (cluster). Clustered messages are received (from the initiating local thread of control (LTC) only) by the `ether-complete-p()` primitive which acts as following: (a) the cluster is read and saved as a Lisp list (*c-list*), and (b) the `car` of *c-list* representing its length is returned as a value. Any subsequent application of `listen-ether()` returns the current `car` of $c-list$ as a message sent and received in the ordinary way, i.e. via network. It should be mentioned that BMP has been developed in C. A Lisp approach (our first attempt) was easier to implement but it requires either a specific routine for reading clustered messages, or the implementation of new versions of the built-in *push* and *listen* primitives; apparently, the first solution is not transparent because messages are explicitly treated as ordinary or clustered ones, whilst the second approach is not efficient since generic operations built on top of LISP are less

```
(defvar *lock* ...)          ; Batch Mode descriptive message.
(defvar *unlock* ...)        ; Standard Mode descriptive message.

(defun server ()             ; Remote thread of control.
  (loop READ-MESSAGE-TAG
        (let ((next-msg (listen-ether))) ; Read next message.
        (cond ((equal next-msg *lock*)
               (ether-lock-output)       ; Enter in BATCH Mode.
               (go READ-MESSAGE-TAG))
              ((equal next-msg *unlock*) ; Enter in STANDARD Mode and
               (ether-unlock-output)     ; transmit a clustered message.
               (go READ-MESSAGE-TAG))
              (t (push-ether (manipulate next-msg)))))))) ; Process a data message.

(defun main (arg1 ... argN) ; Local (starting) thread of control.
  ... ; Segment handling ordinary messages.
  (loop
    (push-ether next-msg1 eth1) ; Disperse calculations among multiple RTC's.
    (push-ether next-msg2 eth2)
    INSPECT-CHANNELS-TAG
    (let ((load1 (ether-complete-p eth1 :block t)) ; Read cluster.
          (load2 (ether-complete-p eth2 :block t)))
      (cond ((and (null load1) (null load2)) ; RTC's are still processing, so
             ... Do some processing locally ...
             (go INSPECT-CHANNELS-TAG))
            ((> load1 load2))
             (manipulate (listen-ether eth1)) ; Read clustered messages as ordinary ones.
             ... Schedule properly (1) ...)
            ((<= load1 load2)
             ... Schedule properly (2) ...)
            ((> (- load1 load2) *load-level*) ; When RTC's are very unbalanced.
             ... Schedule properly (3) ...)
            (t ... Schedule properly (4) ...)) (go INSPECT-CHANNELS-TAG)))
```

Figure 6-1: Scheduling processes via Batch Message Passing (BMP).

efficient than their C counterparts.

Figure 6-1 shows the generic manner that multiple RTC's are scheduled when BMP is in effect. The astute reader notices from the code that an RTC enters into the *batch* or *standard* mode whenever it receives the proper message from the scheduler. Thus, at any given time the work-loads of all or some RTC's are available to the scheduler when the later forces the RTC's to enter into *standard* mode; that is, the desired message clusters are read and the proper actions are performed. In particular when the work-loads, i.e. the number of messages processed so far, of both RTC's in the example differ the scheduling stages (2) and (3) would send more messages to the lightest loaded RTC. Stages (4) and (5) would perform the proper actions when the system is very unbalanced. The blocking or not way of reading clustered messages provides the scheduler, as stage (1) shows, with the possibility of performing part of the entire calculations whenever the examined RTC's are in progress.

Recall that scheduling is rapid since the decisions are made immediately after reading only the number of the remotely processed messages (which are pending locally).

```
; Standard code.
(defun std-remote ()
 (loop (let ((next (listen-ether)))
         (when (< next 0)
               (return-from std-remote "DONE"))
         (when (oddp next)
               (push-ether next)))))

(defun std-local (num)
  (let ((eth (car *ether-list*)))
    (dotimes (i num)
      (push-ether (random i) eth))
   (push-ether -1 eth)
   (loop (let ((result (listen-ether eth)))
           (if (equal result "DONE")
               (return-from std-local t)
               (print result))))))
```

```
; Batch (BMP) code.
(defun batch-remote ()
  (ether-lock-output)
  (loop (let ((next (listen-ether)))
          (when (< next 0)
                (ether-unlock-output)
                (return-from batch-remote t))
          (when (oddp next)
                (push-ether next)))))

(defun batch-local (num)
  (let ((eth (car *ether-list*)))
    (dotimes (i num)
      (push-ether (random i) eth))
    (push-ether -1 eth)
    (dotimes (j (ether-complete-p eth))
      (print (listen-ether eth)))))
```

Figure 6-2: Demonstrating the Batch Message Passing (BMP) concept.

BMP also contributes to the faster execution of an application since each clustered message replaces multiple transmissions to a single one. This is very important because the overhead due to acquisition for only initiating the sending or receiving of a message is of an order of milliseconds. We measured the performance in speed due to BMP by applying the experiment stated in figure 6-2. According to this experiment a randomly constructed numeral sequence is transmitted element by element to a remote process which returns all numbers that are odd. The ping-pong-like problem is solved by an ordinary ETHERLISP program labeled as standard code, and a second one which employs BMP. Note that the same exactly in number input/output messages flow in both programs. The experiment showed that for sequences of 32, 64, 128, and 256 elements the BMP code was 37.5%, 44.5%, 21.1%, and 19.8% faster. We observe that while the length of the list becomes larger BMP slows down. This is caused by the expensive run-time handling of list objects; the same behaviour is also observed when FILOS decodes large lists (see table 3.3). In general, the construction of lists by appending objects is more expensive than the alternative (and adopted) method of *cons*ing objects and reversing the final list. A particular to this case experiment revealed that the construction of numeral sequences of 64, 128, 256, 512, and 1024 elements was approximately 2, 7, 7, 9, and 13 times faster when *cons*ing objects. Remarkable is also the manner remote results are collected. The standard solution utilizes

a loop which breaks when the (additional) message "DONE" is received. This means that for n messages n additional comparisons are performed. Conversely, the BMP solution utilizes a simpler loop with predetermined iterations that definitely completes faster.

6.3 PRAXIS: A Paradigm for Parallel Execution of Instances

Several paradigms have branded the field of concurrent processing with their own novel features. Indicatively, the TIME WARP paradigm [Jef85] guarantees that co-processes never block; in FUTURES [Halst85] co-tasks, called *futures*, can immediately proceed to their computations even if data is not available at the time of their creation but it will be in the future (in some cases though, otherwise futures are suspended). According to our opinion LINDA [NCDG89] is the most attractive paradigm because its particular operations are few, clear, and well defined to produce compact and efficient parallel algorithms. However, we believe that the main drawback of the above paradigms is a significant loss of the overall performance due to their complex and time-consuming underlying handling mechanisms. TIME WARP suffers from large overheads when processes executing erroneously far ahead from a periodically estimated global time stamp are rolled-back to a past (safe) time; Hutchinson [DHJF90] mentions that TIME WARP is of benefit when co-processes need to synchronize relatively infrequently and when normally make progress at approximately the same rate, which are the conditions the incidence of rollback is low. FUTURES usually require a vast number of very fine-grained tasks to be created at run-time resulting in expensive scheduling [Yuen90]. Finally, LINDA also suffers from heavy overheads caused by scheduling multiple mutually exclusive accesses to a shared and distributed tuple space.

PRAXIS is a paradigm that allows concurrent execution via *procedural entities* which are any valid Lisp functions, macros, or lambda expressions. The bodies of procedural entities can be thought of as black boxes that process their input in the sense of actual parameters passed from the sender's lexical environment, and produce an output to the receiver's environment. Procedural entities execute sequentially by means that their bodies do not include any communication primitives. Communication and synchronization among memory-disjoint processes is achieved via messages, called *praxis*, whose structure is:

Praxis ⟶ invoke <operator, operant(s), directive | condition>

The *operator* component is the symbolic name of a procedural entity, and the *operants* are an arbitrary number of parameters applied on the corresponding operator upon re-

ceipt of a *praxis*. The last component denotes either a *directive* or a *condition* which acts as a protocol specifying the manner a procedural entity is to be applied and scheduled in the receiving address space. Figure 6-3 demonstrates the complete set of directives

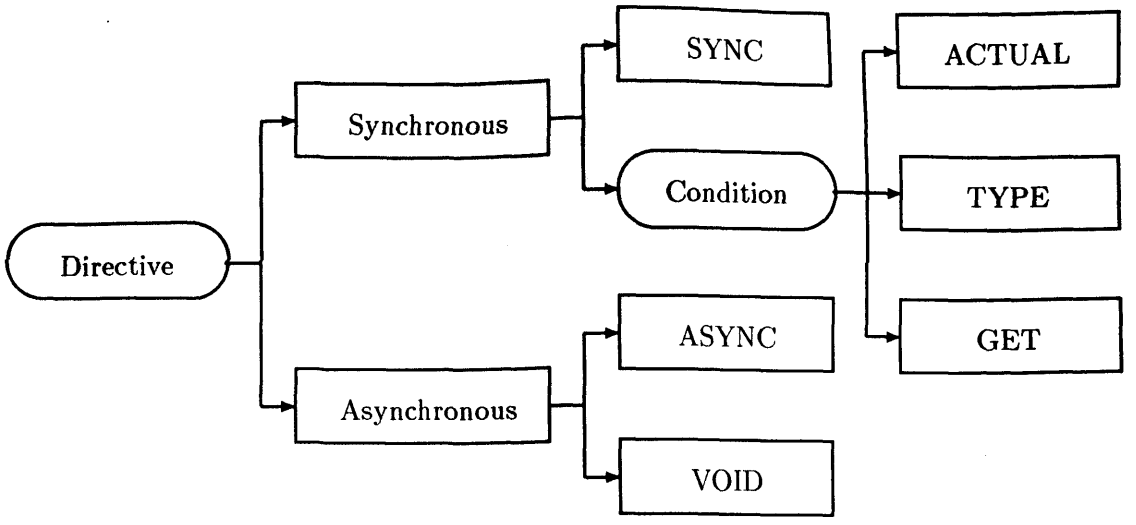


Figure 6-3: Procedure-Invocation Condition-Synchronization Message Passing (PICS/MP).

and conditions called *Procedure-Invocation, Conditional-Synchronization Message Passing (PICS/MP)*. There are two basic directives, SYNC and ASYNC, that provide synchronous and asynchronous message passing respectively. In particular, the ASYNC directive denotes that the sender immediately resumes execution after the sending of a *praxis*, whilst a potential reply can be received at a future time. The VOID directive is used in cases the sender does not require any response from the receiver of the *praxis*. Conversely, the SYNC directive denotes that the sender is always blocked until the receipt of a reply message, in fact another procedural entity.

The fundamental feature of PICS/MP is the applied conditions when the SYNC directive is in effect. The ACTUAL condition denotes that the sender is blocked until the evaluation of the transported operant (in the receiver's address space) yields as a result a Lisp object identical to the one specified by the condition. The TYPE condition stands as a generalization of the ACTUAL one; this condition is satisfied when the operant yields a Lisp object of type identical to the specified one. The last condition GET is used for transferring data between two interacting processes. In particular, GET indicates a place-holder which is assigned a Lisp expression after the sent *praxis* has been evaluated.

6.3.1 Synchronizing Processes in PRAXIS

For a deeper understanding of the PICS/MP concept the problem of matrix-by-matrix multiplication, as it has been defined in section 4.4.1, is considered. According to the PRAXIS's theory the problem must be decomposed into the proper number of procedural entities. The number and the activities of every procedural entity are determined basically by the manner in which (shared) data structures are to be dispersed across network. An initial point could be the remark that any algorithm encountering this problem requires the matrices being multiplied passed as its actual parameters; therefore the dimensions of the result product matrix can be calculated prior to the multiplication. As an effect the order particular elements of the product matrix are to be generated can be defined prior to any calculations too, and hence each co-process can receive the proper data. In our implementation this ordering is achieved by the current value of the global counter `*next-product-count*` (initially set to 1). Additionally another two global variables are needed: the `*last-product*` indicates the numerical value of the last product (64 in the 8×8 case), whilst `*product*` specifies the result matrix (initially all elements are set to `nil`). The solution proposed assumes that each processor is assigned one worker-process and all processes are scheduled by a central manager; moreover, `*dim*` indicates the known dimensions of the input matrices.

The second step of the analysis requires the definition of the address space in which several procedural entities will perform. This step has a tight relation with the complexity of the problem and the granularity of the potentially extracted parallelism. In the simplest case it would be desirable for each remote worker-process to generate a particular product. Thus the corresponding procedural entity could be defined as:

```
(praxis!deffentity compute-product (i j row column)
  (let ((row-len (length row)) ; Rows and columns are of the same length.
        (prod 0))
    (dotimes (q row-len) (incf prod (* (aref row q) (aref column q))))
    (list i j prod)))
```

Note that (currently) the symbolic name `praxis!deffentity` is a nickname of the built-in primitive `defun()`, and its primary aim is the distinction among procedural entities and ordinary Lisp procedural blocks. Although the generation of a product requires only a row from matrix A and a column form matrix B, the position i of a row in A and j of a column in B is also included among the operants for reasons being explained below. The procedural entity `compute-product()` as defined can serve as the operator of a *praxis* message while

its parameters are the operants. For now note that the evaluation of its body returns a list including a generated product and its coordinates (position) in ***product***.

A question arises on how data are first dispersed and second made available at multiple remote sites. This can be achieved if a remote process requests the evaluation of the appropriate *praxis* in the address space in which all data structures are defined. The operator of such a *praxis* could be defined as the following procedural entity:

```
(praxis!deffentity get-next-product ()
  (if (> (incf *next-product-count*) *last-product*)
      nil ; All products have been dispersed.
      (let ((i (floor (/ (1- *next-product-count*) *dim*))) ; Row's position.
            (j (mod (1- *next-product-count*) *dim*)))      ; Column's position.
        (list 'compute-product i j (get-row i) (get-column j)))))
```

We observe that operators with no operants are valid components of a *praxis*. The sequential (non-interrupted) evaluation of its body yields the construction of the procedural entity `compute-product()` outlined above. Note that all processes have been initialized and all (function) definitions, except the input matrices, are globally accessible. The serial order of the next product is designated by incrementing the `*next-product-count*` counter; when its value exceeds the value of `*last-product*` a `nil` is returned signaling the receiver that its mission has been completed as we shall see a little below. For example, when the dimensions of the input matrices are 16×16 then the dimensions of `*product*` are also 16×16; the returned result of this *praxis* for the, say, 2nd product (serial order 2) is the procedural entity (expression): `(compute-product 0 1 #(...) #(...))`, where rows and columns are represented as simple vectors.

The execution cycle of each remote worker process performs the actions specified in the body of the following function:

```
(defun remote-worker ()
  (loop (praxis!invoke (praxis 'get-next-product) :get '?expr)
        (let ((result (eval expr)))
          (if result ; If not NIL then request the next product.
              (praxis!invoke (praxis 'set-product result) :void t)
              (return t))))
  (praxis!invoke (praxis 'praxis!complete) :void t)) ; Inform the scheduler.
```

The condition GET has as an effect the blocking of the (remote) sender until the *praxis* `(get-next-product)` is evaluated (locally by the scheduler), in which case the place-holder `?expr` is assigned with the expression produced by `get-next-product()`. Actually, `expr` is another procedural entity whose (remote) evaluation yields either a product (result) or `nil` indicating the termination of the infinite loop.

When the product has been calculated it must be sent to the address space in which ***product*** is defined. This is accomplished by the procedural entity whose definition is:

```
(praxis!deffentity set-product (i j prod) (setf (aref *product* i j) prod))
```

Now it is clear why the coordinates of a product are included in most of the previous operants. The evaluation of `set-product()` automatically locates `prod` at its correct position in `*product*` according to the values of `i` and `j`. Since no reply is needed the condition VOID is utilized. Note that the VOID condition does not block the sender which immediately exports the next *praxis* requesting another product. For now `praxis!complete()` is a system *praxis* used to terminated a process as it will be described in section 6.3.5.

6.3.2 The Asynchronous Approach

PRAXIS provides an asynchronous processing scheme when the ASYNC directive of PICS/MP in figure 6-3 is employed. In this case algorithms in PRAXIS are designed similarly as in ETHERLISP (see chapter 4) since total control is performed explicitly by the single initiating processor, instead of a transparent scheduler. As a point of departure for asynchronous processing in PRAXIS is the provision of an alternative way for expressing parallelism that some users might consider it more suitable for them. Although the programming style is changed when thinking asynchronously the philosophy of PRAXIS would be entirely preserved. This is apparent in figure 6-4 where the problem of matrix-by-matrix multiplication is solved asynchronously. The algorithm utilizes all of the procedural entities presented in the previous section, except that no specific hook function (like `remote-worker()` above) that controls *praxis* messages remotely is required. This is due to the nature of the ETHERLISP's remote threads of control (RTCs) which are complete Lisp evaluators; their processing cycle includes the continuous evaluation of any received message and the automatic transmission of the generated result(s) back to its sender. Note that while an RTC evaluates a message multiple messages can be accepted and queued in a FIFO order simultaneously. Therefore, messages can encapsulate plain procedural entities, e.g. not *praxis* entities with included conditions, transported to multiple RTCs and evaluated in parallel.

The primary difference of the ASYNC directive is that crucial points such as the collection of remote results are performed in a more compact and abstract fashion, as well as in many alternative ways suitable for a broad range of particular algorithms or network traffic conditions. Meanwhile, an asynchronous approach is more conservative in resources; the algorithm of figure 6-4 uses only two processors but the corresponding synchronous one

```
(praxis!deffentity get-next-product () ... )
(praxis!deffentity compute-product () ... )
(praxis!deffentity set-product () ... )

(defun praxis-async-mm ()    ; Executes on the (local) initiating processor.
  (praxis!declare :async t) ; Switch PRAXIS into asynchronous mode.
  (loop (praxis!invoke (praxis 'get-next-product) :async t) ; Remote invocation.
        (let ((result (praxis!local 'get-next-product)))    ; Local invocation.
          (if result ; If NIL then all products have been generated.
              (praxis!local 'set-product result) ; Set a locally generated product.
              (return t))))                      ; End of the first algorithm's portion.
  (praxis!exec/par (/ (* *dim* *dim*) 2) 'set-product)) ; Collect and set remote results.
```

Figure 6-4: Matrix-by-matrix multiplication: An asynchronous solution in PRAXIS.

requires an additional processor for the central scheduler. Moreover explicit termination of remote processes is not performed; after the evaluation of the last message an RTC either evaluates the next queued message, or it blocks until the arrival of the next one. The algorithm of figure 6-4 splits the problem in two equivalent in processing requirements halves each performing in a different address space (locally and remotely). The careful reader may have observed that the algorithm consists of two logically separated portions; the first portion generates the products as a series of subsequent evaluations of the procedural entity `get-next-product()`, and the second one is devoted only on collecting remote results. More precisely, when the first portion terminates all of the products have been generated but only half of them have been located in the result matrix `*product*`. The half remaining (remotely generated) products are collected and located separately (last line in code). This approach may be very efficient if the network is very loaded; the (local) sender immediately resumes execution after the sending of a *praxis* which is translated as the immediate calculation of the next product locally. When the process for collecting results starts the (one in the example) RTC may still generate products, whilst numerous *praxis* messages may still be pending; but it is highly probable (though certain) that some results have already been transmitted. It is also highly probable that before the collection process terminates the RTC will have finished evaluating all of its queued messages. However, this approach is not safe especially for algorithms that generate a large number of results, because only a limited number of messages can be queued. Besides this limit becomes even smaller as the messages' byte-size grows. Evading for the moment the problem of overflowed message queues we believe that in many cases of algorithms with a known and small number of interchanged messages of any byte-size this strategy can be applied; for vast or unpredictable

number of generated result messages a safe solution is the following:

```
(defun praxis-async-mm ()
  (praxis!declare :async t)
  (loop (praxis!invoke (praxis 'get-next-product) :async t) ; Remote invocation.
        (let ((result (praxis!local 'get-next-product)))    ; Local invocation.
          (if result ; If NIL then all products have been generated.
              (progn (praxis!local 'set-product result)          ; Set local product.
                     (praxis!local 'set-product (praxis!value))) ; Set remote product.
              (return t)))))
```

This algorithm performs as the previous one but each cycle completes with the settlement of two concurrently generated products in the result matrix. Note that `praxis!value()` returns a remote result from an arbitrarily selected communication channel (in case of many RTCs) since the order that products are located in the result matrix makes no difference. For instance, if the RTCs A and B is to generate the products with coordinates (0,0) and (0,3) respectively, and assuming that A is very loaded then the selected B's channel will yield the settlement of the (0,3) product first which is perfectly legal. Obviously, long network delays can restrict the overall performance of this algorithm in terms of delaying the algorithm's portion dealing with local calculations. To overcome this problem another technique can be applied according to which a result can be collected only if one is available. This is achievable if the sixth line of the above code is replaced by:

```
(loop ... (let ((remote-result (praxis!value :block nil)))
            (when remote-result (praxis!local 'set-product remote-result)
                                (incf remote-result-count))) ... )
; Collect any remaining results minus all potentially collected above.
(praxis!exec/par (- (/ (* *dim* *dim*) 2) remote-result-count) 'set-product)
```

From the performance point of view this solution stands in between the two last alternative approaches whilst it seems safe. Frequent bidirectional communications (usually) means that messages are generated (transmitted) and evaluated within short time intervals; hence, even when there are long network delays a frequent channel inspection can serve a large number of pending result messages. Experiments showed that under normal network conditions only the 3%, 5%, and 1% of 32, 128, and 512 result messages are served by `praxis!exec/par()` in the last code fragment. Finally, a totally safe as well as efficient approach is to collect n results after n *praxis* messages have been sent.

6.3.3 Performance Measurements

The matrix-by-matrix multiplication problem is not the ideal algorithm for measuring the performance of a network-based concurrent system for reasons already explained in sec-

tion 4.4.2. Though improper it has been chosen as the basic vehicle for measuring the performance of PRAXIS because it is the most opportune for illustrating many of the features of the paradigm. The second and third columns of table 6.1 show the measurements achieved by the standard sequential and parallel algorithms presented in section 4.4.1. The rest columns state the measurements when the problem is encountered according to the semantics of PRAXIS; thus, the columns labeled as *SEQ*, *ASYNC*, and *SYNC* correspond to the running times measured from a sequential (page 110), and the asynchronous and synchronous algorithms illustrated in section 6.3.1 and figure 6-4 respectively. The first remark is that PRAXIS is slower than ETHERLISP or even than LISP but it is faster than the PRAXIS sequential algorithm. This is due to the larger overhead of the *SEQ* algorithm as the second and fourth columns denote. On focusing on PRAXIS' measurements one may observe that calculations complete faster when they are applied asynchronously. Though the difference in speed could be considered negligible it does exist because the (and any) synchronous algorithm utilizes an additional processor for the scheduler. Note that in all parallel algorithms two co-processors devoted on calculating products are utilized; since concurrency on a single processor is not currently allowed, the scheduler occupies a devoted (additional) processor. Hence, approximately the double in number messages flow bidirectionally in the *SYNC* case. For example, for 2304 products (48 × 48 input matrices) the co-processes in both ETHERLISP and *ASYNC* algorithms exchange 2304 (bidirectional) messages, but in the *SYNC* case the scheduler handles 3456 messages. Therefore, the significantly increased communications burden has as an immediate effect an increased, very small though, overall *SEQ* running time. This is very important because the evidence suggests that *praxis* messages are served (and evaluated) in a more effective manner synchronously denoting that process synchronization via PICS/MP is efficient. This would be justified partly as a better application of parallelism synchronously, and partly due to PRAXIS's scheduling policy (see section 6.3.6); considering short network delays it is highly probable that two remote worker-processes receive and hence evaluate the procedural entity `(calculate-product i j #(...) #(...))` at approximately the same time (perhaps too often). Conversely, in the ETHERLISP and *ASYNC* algorithms any delay on receiving a message suspends the entire algorithm, or more precisely the portion handling the (next) local calculation.

The last two lines of table 6.1 state the measurements of the same experiment except that the communications cost has been artificially isolated. The procedural entity responsible for the calculation of a product has been delayed for three seconds; thus, assuming normal

network conditions potential delays due to sporadic network traffic have been completely eliminated. Under these circumstances all parallel algorithms complete within the half of the sequential running time. This is the expected result since in all parallel cases half of the problem (the remote) is completed simultaneously with the other half (the local).

Test	LISP	ETHERLISP	PRAXIS		
			SEQ	*ASYNC*	*SYNC*
4 × 4	0.950	0.400	0.567	0.517	0.733
8 × 8	2.858	1.933	4.467	3.233	3.817
16 × 16	15.166	12.600	24.633	21.783	21.850
32 × 32	139.338	89.567	250.217	194.800	189.800
48 × 48	391.994	279.000	626.783	449.850	441.333
4 × 4	32.300	16.267	32.553	16.517	16.667
8 × 8	129.983	65.517	132.617	67.350	67.283

Table 6.1: Matrix-by-matrix multiplication: Comparative results between PRAXIS, LISP, and ETHERLISP (time measured in seconds).

The fact that the (synchronous) PRAXIS paradigm is influenced from significant network delays, positively in some sense, is also shown on table 6.2. This experiment was carried out at midday (peak hours of network traffic and host work-load) on Sun 3/60's workstations - all other experiments were exclusively performed on less powerful Sun 3/75's the only available at that time period. The PRAXIS' performance is compared with the one extracted from the same standard algorithms of the second and third columns of table 6.1. According to these measurements the PRAXIS's performance is poor comparatively with the other two algorithms but its performance increases dramatically as the size of the problem grows. More specifically, for very small matrix sizes PRAXIS is 6.5 times slower than the sequential approach whilst for 64 × 64 matrix dimensions it is 1.1 times faster. Conversely, ETHERLISP yields a slightly better but constant speedup in any case of the experiment; this is caused by the non-instantaneous ordered receipt of a large in number results. Apparently, this delay is very low in synchronous PRAXIS because results are collected and processed only when they are available and in any order.

Important is also the reference of the ways PRAXIS' performance can be improved. In general this can be achieved either by increasing the number of co-processors, or by increasing the work-load encapsulated within single transported *praxis* messages. Both enchancements have been adopted and illustrated on table 6.3. The measurements of the table are grouped as two separated modules; both modules employ up to four co-processors (distinct worker

Test	LISP	ETHERLISP		PRAXIS	
4 × 4	0.343	0.328	1.1	2.238	-6.5
8 × 8	2.366	2.017	1.2	6.617	-2.8
16 × 16	16.077	15.166	1.1	25.483	-1.6
32 × 32	118.044	101.433	1.2	136.967	-1.2
64 × 64	996.183	759.750	1.3	969.033	1.1

Table 6.2: Matrix-by-matrix multiplication: Comparative results between LISP, ETHERLISP, and Synchronous PRAXIS (time measured in seconds).

processes) but the procedural entity **compute-product()** generates two adjacent products in the first module, whilst in the second one generates four products. The measured running times are compared with the ones produced by the standard sequential algorithm (second column of table 6.1). Recall also that our interest is focused on the synchronous branch of PICS/MP (the most important) so an additional processor is devoted only for the central scheduler. One may observe that PRAXIS is slower for two products per *praxis* message and up to three worker processes (first module). The performance increases and the paradigm yields faster running times in all cases of the second module; without loss of generality we can consider the measurements for the 4 × 4 matrices as inaccurate or improper for network-based systems. Despite the scheduler's increased overhead due to multiplexing/demultiplexing *praxis* messages the significantly reduced number of messages and the coarser granularity of the algorithm results to a further speedup. For instance, for three workers and four products per *praxis* message PRAXIS is 22.2% faster. Apparently, the performance would be much higher if instead of the fast sequential algorithm the *SEQ* one described above had been used for comparison (see table 6.1). Finally, PRAXIS starts performing less well for four processors because the overhead due to serving multiple communication channels along with the increased communications cost supplant the reduced processing time.

		Two products per *praxis*			Four products per *praxis*		
Test	LISP	2+1	3+1	4+1	2+1	3+1	4+1
4 × 4	0.350	0.617	1.117	0.783	0.417	0.500	0.600
8 × 8	2.394	2.767	2.617	2.167	1.983	1.967	1.983
16 × 16	16.917	18.467	18.100	17.700	12.883	13.550	13.983
32 × 32	123.067	127.433	119.250	119.117	98.833	95.817	100.567

Table 6.3: Measuring PRAXIS performance when multiple co-processors are emploied.

In general lines the above experiments showed that the performance of synchronous PRAXIS is not its major advantage. Of course this particular example algorithm is not the absolute one for drawing definitive general remarks. PRAXIS has been conceived recently and hence it was impossible to make the implementation highly efficient in absolute terms; besides, our particular interest was mainly focused on a particularly flavoured manner disjoined-memory processes can communicate and synchronize - this will become evident in section 6.3.6 where the dining philosophers problem is examined. However, synchronous PRAXIS could be also applied in a shared-memory environment with potentially much better performance due to the non-existent communications cost - this subject is covered in details in the next chapter.

6.3.4 Simplicity and Expressiveness

The extended study of the matrix-by-matrix algorithm brought into the light some of the (mostly synchronous) PRAXIS' special attributes; the most important include high degrees of transparency, scalability, and low scheduling burden. TIME WARP is not considered highly transparent because the sender must explicitly specify the receiver. In PRAXIS messages are anonymous since the primary goal is (or would be) the accomplishment of multiple homogeneous computations and not who exactly accomplish them. Likewise it is of no interest if one processor computes more calculations than another because the fastest ones are granted by the scheduler; indeed, processes influenced by particular network links with heavy traffic or particular overloaded hosts, simply carry out less bulk of computations because other processes may complete the cycle receive-evaluate-send of a *praxis* more frequently. Strictly speaking a potential lack of work-load equilibrium among co-processes in PRAXIS would not affect the overall performance of an algorithm. This remark is of great importance because TIME WARP for example is extremely vulnerable when processes are imbalanced; the balance would be reestablished, if heavy-loaded processes migrate along with their entire contexts to less overloaded hosts [CBJM89]; this particularly expensive strategy also requires additional overheads for a periodically estimation of the load of all participating hosts, as well as the network-wide informing of the new IP addresses of the migrated processes.

The loose relation among self-identifying and independent procedural entities entails the development of scalable algorithms; an algorithm working with two processes will do so with any number. The easiness of converting a sequential algorithm into its parallel corresponding was remarkable. For example, in the matrix-by-matrix multiplication case all

procedural entities (but treated as ordinary functions) were used intact whilst control was achieved by the initiating function defined as:

```
(defun seq-praxis-mm () ; This is the SEQ algorithm of table 6.1.
  (loop (let ((result (praxis!local 'get-next-product)))
          (if result (praxis!local 'set-product result)
                     (return t)))))
```

Such an effortless conversion with no modifications and insignificant additions we have observed only in the FUTURES paradigm. The process of debugging was also proved extremely simplified. Procedural entities yield their output after their sequential bodies have been evaluated. Therefore, concurrency can be simulated by simply passing the output of one procedural entity to another, as usually is the case, manually; when it has been made sure that this scheme works on a single processor then information passing can be (safely) performed via communication channels. Vital is also the easiness existing algorithms can be improved. In the case of the sample algorithm all procedural entities needed minor modifications (mainly an increased number of arguments) to calculate, say, four or more products at once. Note that in this case the transferred bytes are significantly reduced; only one row and four columns are included in a *praxis* message for the production of four successive products. Finally, the asynchronous approach allows many alternative ways of collecting results that at least one will be the most suitable to a particular case.

6.3.5 PRAXIS versus LINDA and TIME WARP

The above example problem has important implications when it is compared with a similar LINDA algorithm. The comparison is theoretical since any attempt of implementing a counterpart algorithm in ETHERLINDA (section 5.6) and measuring its performance will be misled. Basically, ETHERLINDA has been developed for experiencing with both ETHERLISP and LINDA's special features. The underlying LINDA's S/Net kernel is [1] machine-dependent and beyond the scope of this research.

The LINDA approach of solving the matrix-by-matrix multiplication problem given in [NCDG86] employs an additional process for initializing the (shared) tuple space (TS). Prior to creating and initiating the predetermined number of worker processes the TS must contain a series of *out*ed tuples specifying all individual rows and columns of the input matrices. The initialization phase terminates when the special *Dot* tuple is inserted into TS; *Dot*

[1] The examined LINDA implementation [NCDG86] is based on the S/Net machine which consists of numerous MC-68000 processors connected via a word-parallel bus of 80Mbit/sec capacity.

serves as a unique token, as in the ring network technology, indicating the next product being calculated. The next product is determined when one worker (eventually) succeeds to remove (grab) *Dot* from TS and increment the (globally unique) value of the encapsulated counter. Another process is also required for collecting result tuples from multiple remote TS's which contain the calculated products. Thus, the execution time of the LINDA algorithm is divided into processing split among the (parallelly executing) workers, and processing that is inherently sequential. The later sort of processing is absent from PRAXIS because initialization is not required (no TS), whilst collection of results is an inseparable part of the overall (concurrent) processing. The (shared) input matrices are handled by the scheduler which disperses data when serving (independent) requests of the ***praxis*** `get-next-product()`; results are simultaneously collected and processed when serving the instructive *praxis* message `set-product()`. Inherently sequential processing in LINDA is also considered the time spent in S/Net kernel interrupts. If several workers attempt to remove a tuple simultaneously then only one (system-widely) must succeed. This can be achieved by a particular *delete protocol* which according to the original implementors its (enchanced) activities include the following: The operation *in(s)* entails the broadcast of the template s to all network nodes. Upon receipt a node matches s in its local TS; if there is a match the matched tuple is transmitted back to the (kernel of the) requesting node; otherwise s is stored for x time units and then throws it out (in the mean time all tuples generated within this period are checked against s). If the requesting node has not received any response after x time units s is rebroadcast. Apparently, this protocol is responsible for significant overheads, whilst the entire system becomes less reliable [2] as an effect of the increased probability of a network failure due to additional "invisible" message transmissions. In PRAXIS neither broadcasts are required nor additional transmissions for mutually exclusive operations; the requesting process is simply blocked if the condition in a ***praxis*** can not be satisfied in the receiving address space, or because the monolithic scheduler is serving another process. At this point it should be mentioned the importance of an efficient message handling mechanism in cases like LINDA's delete protocol; in section 5.6.1 we showed that FILOS yields up to 71% excluded byte size of tuples interchanged among trivial LINDA processes. Now we are in position to say that this measurement is higher if we take in account the additional transmissions (of the same tuples) because of the delete protocol.

[2] The implementors of S/Net kernel argue that a negative-acknowledgement signal is available on the S/Net bus when some node fails to receive and buffer a broadcast message. Thus, the delete protocol depends heavily on the unavailable negative broadcast acknowledgement signal on an Ethernet-based network.

Dot is re-inserted into TS if there are remaining products to be calculated; otherwise the tuple is remove from TS causing all workers to be blocked upon trying to read a non-existent tuple. At this point an (additional) clean up process starts collecting results by *in*ing result tuples from all remote TS's. Since all workers are in a spin-lock state requesting the *Dot* tuple the clean up process would be substantially delayed from the delete protocol because it continuously receives and handles additional broadcast messages every x time units. Furthermore, results in LINDA are collected in a determined serial order by two trivial C *for* loops. Thus, a correct solution of the problem imposes either the start of the clean up process when all workers are finished (or blocked) and which is adopted by the examined Linda algorithm, or by suspending the clean up process each time a particular requested result tuple is unavailable (due to a slow worker). In PRAXIS results can be collected and processed in any order and hence network delays or heavily loaded hosts do not affect (substantially) the overall algorithm. Recall that there is not any restriction when the 20th result is positioned in the result matrix while the 10th one has not even been calculated. Moreover, in LINDA a "clean" environment for initiating the same algorithm with different input, or a new problem is not preserved. Blocked workers can be restarted by dropping into TS a new *Dot* tuple. In PRAXIS process termination is performed in a more sophisticated manner and it is transparently carried out as part of the entire processing too. When a remote process is signaled that its mission has been accomplished (see section 6.3.1) the system's *praxis* `praxis!complete()` is sent to the scheduler; its evaluation yields the increment of a PRAXIS' counter which when equals to the (known) number of worker processes scheduling is considered finished. At this point any sort of problem can be started since all remote evaluators (RTC's) are "clean", e.g. blocked awaiting on their input channels the arrival of a (and of any context) *praxis* message.

Assume a large scientific project, say multiplying matrices with thousands of rows and columns, which is expected to complete within few weeks or even months; as a consequence several incidental or not host or network failures are very likely. Assuming a large number of participating hosts, if any host crashes after the *praxis* `set-product()` has been successfully completed there is no side effects in a PRAXIS algorithm; it will carry out with the remaining hosts (in the worst case with a single host). The problem perplexes, slightly though, if a host crashes before or during the calculation of a product; this implies that this particular result will be absent from the result matrix `*product*`. The problem can be easily handled by running a recover-procedure that checks for `nil` elements in the result

matrix; if any they can be calculated locally as in the asynchronous algorithm in figure 6-4. Note that the recover-procedure can be run either for reasons of prejudice when all workers have been completed successfully, or when a broad periodical inspection of channels yields broken network links. The only problem of PRAXIS is when the initiating host crashes, but in such a case ***product*** can be saved periodically and passed as an argument to the above recover-procedure. An efficient LINDA algorithm requires the collection of results after all products have been calculated; thus, a host failure implies the loss of a TS instant and all *outed* in it unique products; additionally, the strict relation between tuples, processes, and TS's imposes a more complicated recover-procedure.

Finally, the longest delays and waste of resources in LINDA are caused by the expensive pattern matching and the replicated TS's. The problem of implementing an efficient tuple matching operation has been noticed by Carriero [NCDG86, p:118], as well as by Leichter [Leich89, p:58] who implemented the commercial VAX LINDA-C language. Both implementations rely on a hashing scheme based on the initial field (tag) of every tuple; this approach suffers from large scattering and hence large searching overheads, especially if the programmer has not chosen the tag producing little scattering or imbalance of the hash table entries. Proposed techniques for a satisfactorily efficient pattern matching are relied either on special hardware regarding that TS is an associative memory (different than the conventional arrays of fixed size words) which should be handled by a Linda machine; or, based on expectations (yielded from programs' analysis) that the applications being developed usually utilize tuples with only few fields, or that formal fields are extremely rare. The problem of tuple matching is more apparent in a network-based implementation which requires replication of TS (costly in memory requirements), since it is the only way to reduce the expensive communications cost and congestion due to a single TS alternative [UDNMcD90]. Clearly, PRAXIS does not suffer from these two problems; *praxis* messages are self-contained entities whose primary goal is the sequential evaluation of the usually concise and independent operators without any restriction on the number of the applied operants. The communications cost depends only on the number of the interchanged *praxis* messages and no additional transmissions for administrative purposes are required.

Synchronization via information included within messages has been also put into practice by the TIME WARP paradigm; but we believe that in case of PRAXIS the synchronization cost is match less expensive. In TIME WARP synchronization information (among others) include the virtual send time (VST) stamp of the sender's local clock, and the virtual re-

ceive time (VRT) stamp denoting the future time the message should be received. Clearly, a central authority is responsible for synchronizing multiple distributed clocks according to the global virtual time (GVT), and to ensure that VRT is greater than VST of every single message sent. Vast bulks of information (including all sent and received messages) representing the state of each co-process usually are saved periodically to be used in case of a clock mismatch. In such a case a process rolls back to a correct in time state and re-runs computations; in the worst case all processes can be rolled back as an a effect of the antimessages sent from the first "problematic" process. We think that in PRAXIS synchronization is extremely simple and the usage of crucial resources such as memory and network are kept minimal. There is no need for just a single additional transmission due to a delete protocol, or the necessity of cancelling network-wide computations already done. Furthermore, neither replicated message queues nor queues of already processed messages are maintained; queued (pending) messages may exist only due to speedy senders.

6.3.6 Synchronizing Simultaneous Access on Shared Data

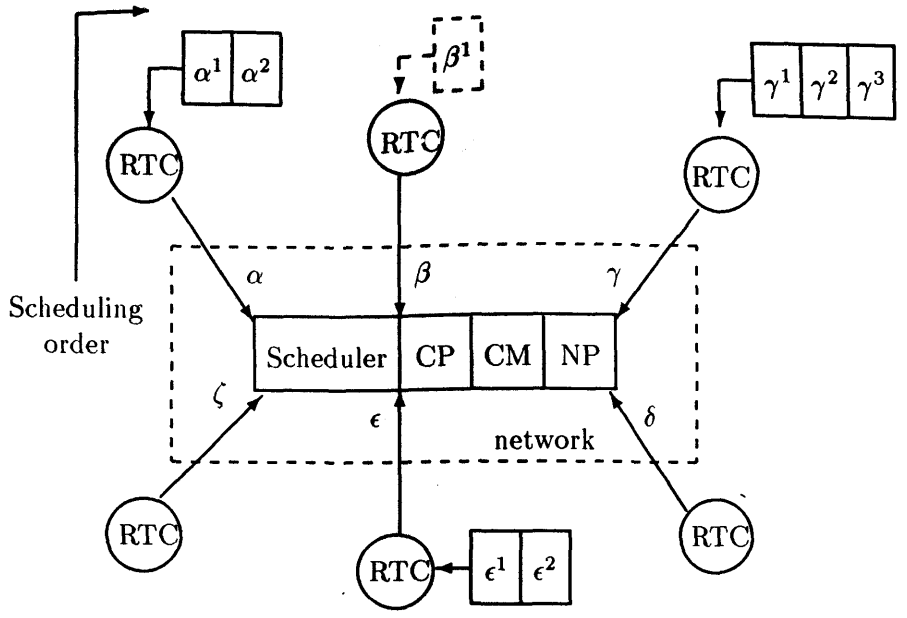
Ringwood [Ring88, p:11] reckons that the dining philosophers problem "... allows the classic pitfalls of concurrent programming to be demonstrated in a graphical situation. It is a benchmark of the expressive power of new primitives of concurrent programming and stands as a challenge to proposers of these languages." We partly agree with Ringwood since in our limited experience we have not seen a simpler solution to this problem as the one in LINDA and PRAXIS both illustrated in figure 6-6. Indicatively, an approximation of code lines for this problem in Parlog86 [Ring88] is 70, in the SOCKET package [3] is 250, in ETHERLISP 150, and in TIME WARP [4] 300 lines.

The synchronous solution in PRAXIS is compact and elegant; meanwhile it is a very good example to illustrate the manner processes synchronize their simultaneous access to shared data. Tickets and chopsticks have been represented, instead of as tuples, as shared data structures in the address space of the initiating (scheduler's) processor. In both solutions only four philosophers are allowed in the refectory at the same time. Upon requesting a ticket a philosopher blocks until the corresponding procedural entity returns an actual `t`. Left or right handed chopsticks are similarly granted according to their numerical value

[3] The SOCKET package has been described in section 5.7 whilst in appendix B we present the definition and solution of the dining philosophers problem in this environment.

[4] This is based on a complete implementation of TIME WARP on top of EULISP [EuLisp93] curried out by colleagues at Bath University.

specified by the ACTUAL condition. The conditions upon invoking a chopstick *praxis* would be also stated as `(... :actual t)` or as `(... :type 'fixnum)`, if the invoked procedural entity had been designed (in one code line) to return a `t` or an integer on success. After a philosopher has finished his lunch he releases all occupied critical resources with no delay as a series of non-blocking *praxis* invocations (with included VOID conditions).



CP = Current Process, CM = Current Message, NP = Next Process

Figure 6-5: The Left-to-Right-Urgency (LRU) scheduling policy of PRAXIS.

As a consequence of the synchronous novel semantics of PRAXIS is a simple, fair, and low overhead scheduling procedure. A concurrent solution of the dining philosophers problem requires six processors; one for each philosopher object and one for the scheduler. Our scheduling policy is based on a continuous Left-to-Right-Urgency (LRU) inspection of all communication channels, which entails a Round-Robin service. The LRU cycle starts from the channel next to the last served one. If there are any pending *praxis* messages on this channel one is peeked and evaluated; if the enclosed condition is satisfied then the *praxis* is removed from the channel's message queue and the connected remote blocked sender resumes execution. The difference between the LRU and Round-Robin policies lies on how the next channel is determined. In LRU the next channel is the one (always in a LRU course) with pending messages; when one is found the next channel, if any, is the one which when was examined had no pending messages. This channel selection scheme is illustrated

in figure 6-5. Assuming that at time t_0 $CP = \alpha, CM = \alpha^1$, and $NP = \beta$ and that channel β has no pending *praxis* at time t_1 but it receives one at time t_2, then the scheduler acts as:

$$t_1 \longrightarrow CP = \gamma,\ CM = \gamma^1,\ NP = \beta$$
$$t_2 \longrightarrow CP = \beta,\ CM = \beta^1,\ NP = \gamma$$
$$t_3 \longrightarrow CP = \gamma,\ CM = \gamma^2,\ NP = \delta$$

Obviously this scheduling scheme is non-blocking assuming normal network conditions and short execution time of *praxis* messages. Furthermore to ensure fast completion of an LRU cycle only one *praxis* message at a time per channel inspection is served. Since messages are first peeked (and not removed) avoids additional overheads for maintaining multiple queues for channels (processes) blocked due to unsatisfied conditions, or processes that were in a processing stage at the time their output channels were inspected.

As a consequence, our scheduling policy gives at a low price high possibilities for serving delayed (due to host or network congestion) processes; meanwhile no single message than the absolutely necessary are transmitted and handled.

Although LINDA yields an elegant solution it suffers at two points from an imperfection and a weakness. First, one might observe that some time is spent sequentially for dropping all sharable tuples into TS, an overhead completely absent from PRAXIS. Second, Carriero in [NCDG89, p:451] mentions that if LINDA's kernel is "unfair" "...the Linda solution allows indefinite overtaking or livelock.". This happens when a slow philosopher (process) is blocked upon *in*ing a ticket tuple whilst a speedy one repeatedly *out*s a ticket tuple and then grabs it again. Assuming very long network delays the same can happen in our solution too. Taking notice that no solution to this problem is proposed by Carriero we may assume that the problem either can not be solved or it requires another more complex (and larger) algorithm. Conversely, in PRAXIS the solution of this "problem" was effortless requiring minor modifications of the existing code. More precisely, the fourth line of `praxis-philosopher()` is replaced by `(praxis!invoke (praxis 'pickup-ticket id i) :actual t)` (see figure 6-6) and the procedural entity `pickup-ticket()` is re-defined as:

```
(defvar *ticket-gap* 3) ; Largest # of rounds without getting a ticket.
(defvar *slowest-phil* (cons 4 *ticket-gap*)) ; Arbitrary selection.
(praxis!deffentity pickup-ticket (id cnt) ; Philosopher's id and # of all granted tickets.
  (when (> *tickets* 0) (if (< cnt (cdr *slowest-phil*))
                            (progn (setq *slowest-phil* (cons id cnt)) (decf *tickets*) t)
                            (if (>= (- cnt (cdr *slow*)) *ticket-gap*)
                                   nil ; Return this and block the speedy process.
                                   (progn (decf *tickets*) t)))))
```

```
;;;;; The EtherLINDA solution.
(defvar *phil-num* 5) ; Number of philosopher objects.
(defvar *rounds* 50)  ; Execute simulation 50 times.

(defun linda-philosopher (id)
  (dotimes (i *rounds*)
    (format t "~%Philosopher ~D is thinking..." id)
    (linda-in (make-tuple "pickup-ticket"))
    (linda-in (make-tuple "pickup-chopstick" id))
    (linda-in (make-tuple "pickup-chopstick" (mod (1+ id) *phil-num*)))
    (format t "~%Philosopher ~D is eating...")
    (linda-out (make-tuple "putdown-ticket"))
    (linda-out (make-tuple "putdown-chopstick" id))
    (linda-out (make-tuple "putdown-chopstick" (mod (1+ id) *phil-num*))))
  (linda-out (make-tuple *linda-zombie* (linda-id))))

(defun linda-init ()
  (dotimes (i *phil-num*)
    (linda-eval 'linda-philosopher i)
    (linda-out (make-tuple "pickup-chopstick" i))
    (when (< i (1- *phil-num*)) (linda-out (make-tuple "pickup-ticket")))))

(linda-schedule 'linda-init)


;;;;; The PRAXIS solution.
(defvar *phil-num* 5)              ; Number of philosopher objects.
(defvar *rounds* 50)               ; Execute simulation 50 times.
(defvar *tickets* (1- *phil-num*)) ; Allow only 4 philosophers in dining room.
(defvar *forks* (make-array *phil-num* :initial-element t))

(praxis!deffentity pickup-ticket () (when (> *tickets* 0) (decf *tickets*) t))

(praxis!deffentity putdown-ticket () (incf *tickets*))

(praxis!deffentity pickup-chopstick (i)
  (let ((ticket (aref *forks* i)))
    (when ticket (setf (aref *forks* i) nil) ticket)))

(praxis!deffentity putdown-chopstick (i) (setf (aref *forks* i) t))

(defun praxis-philosopher (id)
  (dotimes (i *rounds*)
    (format t "~%Philosopher ~D is thinking..." id)
    (praxis!invoke (praxis 'pickup-ticket) :actual t)
    (praxis!invoke (praxis 'pickup-chopstick id) :actual id)
    (praxis!invoke (praxis 'pickup-chopstick (mod (1+ id) *phil-num*)) :actual id)
    (format t "~%Philosopher ~D is eating..." id)
    (praxis!invoke (praxis 'putdown-ticket) :void t)
    (praxis!invoke (praxis 'putdown-chopstick id) :void t)
    (praxis!invoke (praxis 'putdown-chopstick (mod (1+ id) *phil-num*)) :void t))
  (praxis!invoke (praxis 'praxis!complete) :void t))

(defun praxis-init ()
  (dotimes (i *phil-num*) (praxis!eval (praxis 'praxis-philosopher i))))

(praxis!schedule 'praxis-init)
```

Figure 6-6: The dining philosophers solved in ETHERLINDA and PRAXIS.

```
(defvar *phil-num* 5) ; Traditionally five philosophers.
(defvar *rounds* 50)  ; Repeat simulation 50 times.
(defvar *chopsticks* '(t t t t t)) ; All chopsticks are initially available.

(praxis!deffentity pickup-chopstick (id) ; Return T on success, NIL to prevent deadlock.
  (if (>= (count t *chopsticks*) 2) (progn (setq (elt *chopsticks* id) nil) t) nil))

(praxis!deffentity putdown-chopstick (id) (setq (elt *chopsticks* id) t))

(defun praxis-philosopher (id) ; Five (concurrently executing) instances.
  (dotimes (i *rounds*)
    (format t "~%Philosopher ~D is thinking..." id)
    (praxis!invoke (praxis 'pickup-chopstick id) :actual t)
    (praxis!invoke (praxis 'pickup-chopstick (mod (1+ id) *phil-num*)) :actual t)
    (format t "~%Philosopher ~D is eating..." id)
    (praxis!invoke (praxis 'putdown-chopstick id) :void t)
    (praxis!invoke (praxis 'putdown-chopstick (mod (1+ id) *phil-num*)) :void t))
  (praxis!invoke (praxis 'praxis!complete) :void t))
```

Figure 6-7: A prominent solution of the dining philosophers problem in PRAXIS.

We arbitrarily declare the fifth philosopher as the slowest with `*ticket-gap*` tickets in his possession. To pick up a ticket a philosopher must be either slower than the slowest one (in which case he becomes the slowest), or to be no more than `*ticket-gap*` times in the refectory more often than the slowest philosopher. An important issue of the above is that the solution was found and implemented within short time as a result of the PRAXIS philosophy. In LINDA we did not manage to find a solution although a possible one could be the representation of the passive ticket tuple as an active one (using *eval()*); obviously, this entails the creation of a new process executing a code similar to our solution.

The vital point in the dining philosophers problem is the dissuasion of a *deadlock* situation to occur; that is when all, traditionally five, philosophers enter into the refectory and then each grabs his left hand chopstick. One of the most popular ways proposed (and outlined in figure 6-6) for preventing deadlock is to allow only four philosophers in the refectory at any one time. The PRAXIS' philosophy is expressive and powerful enough to exceed the four philosophers boundary. In figure 6-7 we propose a prominent resolution to the problem where all philosophers are safely allowed in the refectory simultaneously. The critical condition which prevents deadlock is that none philosopher is allowed to grab any chopstick unless two chopsticks are available (free) upon request. Thus, in case that the first four philosophers get one chopstick each, the fifth one is suspended and his chopstick is given to the fourth philosopher after a complete LRU scheduling cycle. The PRAXIS scheduling policy eliminates the possibility of a *lockout* situation too; that is when two philosophers,

say 0 and 2, are always granted from two chopsticks each which philosopher 1 never gets. Since tickets in figure's 6-7 code are no longer in use the communication cost is reduced as well, whilst the problem is resolved in an even more compact and elegant fashion.

6.4 Summary

In this chapter we presented two innovative techniques implemented entirely on top of ETHERLISP. The Batch Message Passing (BMP) concept actually is a particular method for handling cluster messages. Its application presupposes multiple generated logically related messages that are not needed immediately and hence, they can be transmitted at a later time as clusters of known length. We showed that clustered messages can yield efficient scheduling schemes, as well as a significant increase in the speedup of a distributed algorithm.
Conversely, PRAXIS is a complete concurrent paradigm. Its semantics are novel, few, simple, and powerful to provide compact, elegant, and in some cases radical solutions to a broad range of problems. The model provides both synchronous and asynchronous approaches on extracting parallelism with the former case as the most flavoured. The fundamental feature of the synchronous approach is the *praxis* message which consists of a particular condition applied immediately after the evaluation of an accompanied procedural entity; procedural entities are complete procedural blocks, usually inexpensive in CPU requirements, contain sequential ordinary code, and describe logically independent portions of an algorithm that can be scheduled for concurrent execution. The loose dependency among *praxis* messages along with the embedded synchronization information results to an efficient and inexpensive scheduling of multiple interacting processes. However, PRAXIS has been conceived recently and hence, it is too early for us to point out exact conclusions with respect to its behaviour and performance. Very efficient algorithms lay claim for particular patterns with which we are not quite familiar; questions also arise about a possible implementation of PRAXIS in a shared-memory environment - discussed in the next chapter - and how efficient it could be.

Chapter 7

Further Research and Concluding Remarks

7.1 Introduction

BASED ON THE SIMPLE STRUCTURE of ETHERLISP and setting several assumptions and limitations we developed an intergrated basic physically distributed system which deals efficiently with numerous real problems. However, during this research we noticed several areas the existing system must be enchanced or should be extended.
In this chapter we discuss all of the enhancements or additions to the existing system that come first in the schedule of future research.

7.2 Extended Process Interconnections in ETHERLISP

The basic structure of ETHERLISP, scheme I in figure 7-1, was kept simple because our primary aim was focused on the achievement of the highest possible performance of a network-based system. Such a connection configuration lays claim for the absolutely necessary communications cost whilst it provides satisfactory flexibility and computation power;

extensions like concurrency within single processes and fully interconnected processors plainly provide additional power and flexibility but turn aside from our current goals.

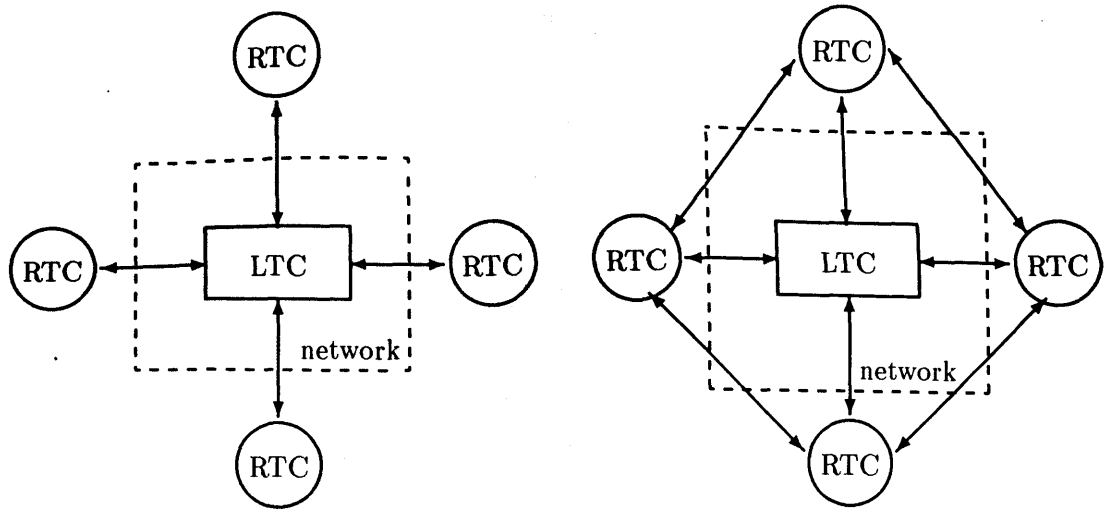


Figure 7-1: Alternative schemes for inter-connecting memory-disjoint processes.

The survey carried out in chapter 4 disclosed a weak point of ETHERLISP. The fact that communication among RTC's is possible only through their common ancestor LTC, in some cases degrades significantly the performance of parallel algorithms. If independent remote communication was possible then some of the remotely generated results could be processed in parallel. More precisely, sorting (see section 4.6) which is a problem with large overheads due to correlating results, the remotely sorted subsequences could be merged remotely as well. This can be achieved by the communication scheme II in figure 7-1. Providing at least three processors this alternative connection scheme would be provided, whenever desirable, by applying:

(**make-ring-ether &rest** *host-name*)

According to the second scheme a unique LTC creates and monitors all RTC's configured as a ring whilst each RTC is allowed to communicate only with LTC and its two neighbouring RTC's in any direction. Obviously, this structure provides greater flexibility than scheme I with less hard and expensive scheduling required by a fully interconnected scheme. Any prematurely completed RTC's can be scheduled either by a local or a remote user-defined handler to cope with partial (and parallel) result correlation. Moreover, debugging, recovery, and overall control can be still encountered relatively effortless.

7.3 Enchancing FILOS

The structure of FILOS as presented in chapter 3 provides a general variable-block method of encoding/decoding and efficaciously prepearing messages being transported through network links. The method is general enough so it can be adopted by a broad range of network-based systems providing minor changes. However, we have located several areas FILOS can or could improve its performance.

First, we have noticed that the delimiters of list objects constitute a considerable amount of the total transferred bytes. Usually Lisp code consists of multiple sublists starting with a single left parenthesis but too often ending by numerous right ones. For example, the 11.8% of the byte-size of the code listed in figure 3-4 corresponds to left and right parentheses. Thus, the right parentheses could be encoded by a special one-byte block denoting the actual encoded information along with the number of adjacent delimiters. This approach yields up to 67.48% (+3.14%) compression performance (excluded byte-size) since 28 right parentheses, e.g. 28 bytes, are encoded only within six bytes.

Second, the characters of symbols and strings could be further compressed, for instance, by the Huffman code [Lelewer87, Capo86]. Apparently, in cases of long character sequences the additional compression will be significant but such an approach will burden the encoding/decoding cycle. During this research we observed that messages usually are short and repeated; thus, a Huffman-like code would only aid in cases of vast amounts of transferred bytes. Note that FILOS almost ensures that all character sequences in the messages of an algorithm are literally encoded once and then remain non-search encoded ATM_i indices.

Finally, the LINDA model and models with similar requirements in constructing and interchanging messages triggered the necessity of a more specialized manner for encoding/decoding and maintaining messages. Although the current kernel of FILOS yields a very good performance in handling Linda tuples (see section 5.6.1), the later can be much further improved based on several properties of tuples. Passive tuples contain either constant templates (place-holders or formals) or variable actuals, but in both cases the first field is a constant actual string or symbol (tag). One approach could be the definition of a *template-tuple* object type with the following structure:

```
struct template_tuple {   /*  Passive tuple (template) structure  */
    short    tt_type;     /*  Implementation dependent type id  */
    object   tt_value;    /*  Any valid Lisp object (here a cons)  */
    short    tt_index;    /*  Index on the TUP-T table  */
};
```

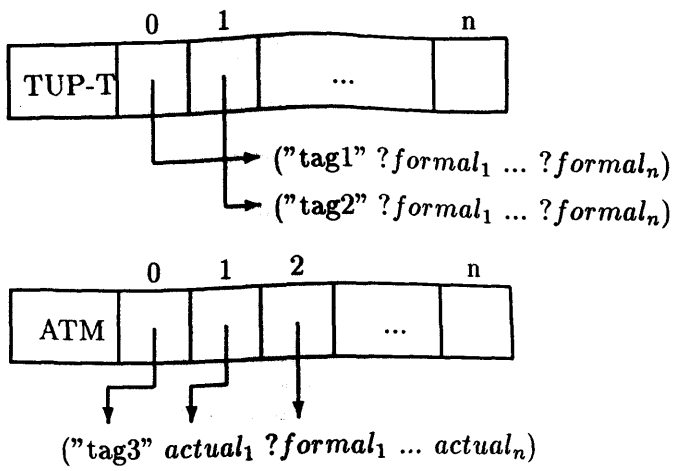


Figure 7-2: Handling of tuple-like messages.

Observing that in many Lisp implementations of LINDA tuples are represented as list objects the second slot points to such a value. Type-identified templates can be recognized by a pre-processing stage which builds the table TUP-T as figure 7-2 illustrates. TUP-T entries are 32-bit pointers to `template-tuple` structures whilst the third slot of the pointed structures holds the corresponding TUP-T pointer values; thus, the searching overhead is nil and pattern-matching is particularly inexpensive since there is no need to encode or decode templates or to transmit more than few bits per template (the index's value) independently from the number or print-name size of the encapsulated formals. Note that TUP-T has a fix size but we assume that a fairly large application usually employs no more than a few hundreds of different templates; meanwhile, memory requirements for storing few 32-bits pointers are considered negligible. In the extreme case that TUP-T runs out of space templates can be handled as the ATM_i table's entries (see section 3.5.3). Finally, tuples with fields which are either all actuals or both actuals and formals can be handled as ordinary ATM_i indices as figure 7-2 illustrates.

7.4 PRAXIS in a Shared-Memory Environment

After their initial network-based implementation of SCHEME LINDA - where TS was handled by a single server process - Dahlen and MacDonald mention that "...Clearly this is not a scalable approach, since the server will very quickly become a bottleneck." [UDNMcD90, p:12]. Although all sample algorithms presented in the previous chapter entail particularly frequent com-

munications - 12296 messages in the 64 x 64 matrix multiplication case - they performed smoothly; but their synchronous solution in PRAXIS is not as efficient as it would be. One reason is that the implementation of PRAXIS on top of ETHERLISP is definitely slower in performance than if it had been implemented as C code. However, we believe that our model would perform better in a shared-memory environment. We reached this conclusion after encountering the problem of sorting.

```
(praxis!deffentity generic-aref (vec idx) ; Accesses the vector's elements via
  (if *praxis!remote*                     ; network when applied remotely.
      (praxis!invoke (praxis 'generic-aref (praxis!verb vec) idx) :sync t)
      (aref vec idx)))

(praxis!deffentity generic-swap (vec elt-x elt-y) ; Swap two vector elements via
  (if *praxis!remote*                             ; network when applied remotely.
      (praxis!invoke (praxis 'generic-swap (praxis!verb vec) elt-x elt-y) :sync t)
      (let ((temp-elt (aref vec elt-x)))
        (setf (aref vec elt-x) (aref vec elt-y))
        (setf (aref vec elt-y) temp-elt))))

(defun partition (vec lb ub) ; Partition the vector in two independent subvectors.
  (let ((mid (generic-aref vec (round (/ (+ lb ub) 2)))) (i lb) (j ub))
    (loop (when (> i j) (return-from partition (cons i j)))
    (loop (if (< (generic-aref vec i) mid) (incf i) (return i)))
    (loop (if (> (generic-aref vec j) mid) (decf j) (return j)))
    (when (<= i j) (generic-swap vec i j) (incf i) (decf j)))))

(defun sort (vec lb ub) ; Sort an independent subvector.
  (if (and (= (- ub lb) 1) (< (generic-aref vec ub) (generic-aref vec lb)))
      (generic-swap vec lb ub)
      (if (> (- ub lb) 1)
          (let ((part-result (partition vec lb ub)))
            (sort vec lb (cdr part-result))
            (sort vec (car part-result) ub))
          (and *praxis!remote*
               (praxis!invoke (praxis 'praxis!complete) :void t)))))

(defun qs (vec) ; Partition the vector and create two worker processes.
  (let* ((lb 0) (ub (1- (length vec)))
         (part-result (partition vec lb ub)))
    (praxis!eval (praxis 'sort praxis!void lb (cdr part-result)))
    (praxis!eval (praxis 'sort praxis!void (car part-result) ub))))

(praxis!schedule 'qs vec) ; Start the system's scheduler.
```

Figure 7-3: Sorting in PRAXIS.

In figure 7-3 the sorting algorithm divides an unordered numeral sequence S in two unordered and of different length subsequences S' and S'', where all elements in S' are smaller than all elements in S''. This division is performed by `partition()` on the starting processor and then each subsequence is assigned to a co-process to be sorted. Clearly, there

is no need for merging the sorted subsequences; the final result is the concatenation of S' and S''. The remote co-processes share access to the elements of S by successively invoking two generic procedural entities; note that these entities distinguish between the local and the remote address spaces. Apparently, the extremely frequent communications cost yields a poor running time as our experiments showed. The evidence suggests that this sorting approach does not perform well synchronously. This of course does not mean that sorting can not be solved in PRAXIS efficiently. According to the algorithms in figures 4-7 and 4-8 in section 4.6 `quick-sort()` and `quick-merge()` can be declared as procedural entities; S is segmented into numerous portions each assigned to a remote process. Apparently, this solution preserves the spirit of PRAXIS is scalable and efficient.

According to Andrews [Andrews91, p:591] the execution times - measured in microseconds on an unloaded Sun SPARCstation 1+ - of different combinations of process interaction mechanisms among others include: 5 μs for a semaphore pair, 105 μs for an asynchronous send/receive pair, and 290 μs for a synchronous send/receive pair. Based on these measurements when the enclosed in procedural entities shared data structures are surrounded by semaphores, the overall performance of PRAXIS would be significantly improved. In addition the message passing concept can be preserved but in a less expensive scheme since the transported data are pointers to the shared memory.

Another possible implementation could be the creation of a process upon invoking a *praxis* as in FUTURES, or a lightweight *thread* as in EULISP [EuLisp93]; in this approach the use of semaphores is also necessary.

7.5 Conclusions

ETHERLISP is a network-based system whose major aspects include a simple structure providing a transparent concurrent environment for efficiently coping with a broad range of problems. This research consists of several stages each dealing with particular but of vital importance features of distributed systems, such as a proper design and implementation, flexibility, inexpensive demand for critical resources, and finally efficient overall performance. Although ETHERLISP can be considered an intergrated concurrent system various aspects relevant to persistent communications, work-load balancing policies, or the provision of finer independent threads of control are not supported; we believe that all these aspects contribute to a safer and more efficient system but their serious consideration is a matter of extended research. However, taking in account the enhancements outlined in this

chapter the concluding remarks of this research are the following:

The first chapter gives a short description of the layers beneath the structure of any network-based system. It should has been made clear that each layer causes considerable overheads owing to handling data packets across a sharable network; thus, data packets must be prepared by segmenting them or assembling them at the sending and receiving sites, data must be carried safely which may imposes multiple retransmissions, and finally, packets must be multiplexed/demultiplexed among irrelevant user domains. Adding the additional overheads for encoding/decoding data messages according to the semantics of a system, network-based systems are open to numerous questions with respect to the amount of parallelism that can be extracted, as well as the effort and the skills either at implementation or programming level required for its accomplishment.

In the second chapter we presented the design and implementation of ETHERLISP according to the stated conjecture; that is, the provision of parallelism through interprocess communications and synchronization. Apart from the simple system's structure, the plethora of well-specified primitives contribute to the extraction of efficient parallelism; it is of our belief that many primitives that perform the absolutely requisite tasks instead of few and generic ones yield minimal overheads for exclusive use of critical resources like the network; obviously, all these adds to one thing which is the better overall system's performance.

The third chapter deals with handling the communication cost which is the most important issue of network-based systems. We paid particular attention on this topic because we envisaged ETHERLISP as a system which is largerly independent from communications; under these circumstances the user is provided with relatively unlimited power, control and flexibility over the way of allocating processes, dispersing data, and making use of frequent or not communications. In general lines this goal can be accomplished by a drastic reduction of the network burden in the sense of compressing data messages, since the costs due to network traffic or network protocols are considered constant, unpredictable, and hence uncontrolled from a distributed system. FILOS aids to this purpose and its structural issues are in accord with the ones of ETHER kernel; that is, FILOS is compact, simplistic, totally transparent, and lays claim for low requirements in resources such as main memory, network usage, encoding/decoding and computation burdens. In chapter 3 all these features showed a highly satisfactory performance of FILOS; the attained compression proved efficient, constant, and independent from the data being manipulated; thus 50% to 65% of excluded data quantity is taken as granted, whilst the overhead produced from its applica-

tion is significantly less than the network's one. In addition, FILOS is a general compression method which can be adopted by any Lisp-like system whilst systems with entirely different semantics can adopt many of its features. However, the importance of an efficacious compression technique becomes apparent in subsequent chapters where FILOS is applied in a more realistic manner; in chapter 4 FILOS is responsible for a large reduction of the overall running time of various real applications; in chapter 5 the LINDA paradigm itself as well as its underlying synchronization mechanisms take advantage largerly from FILOS; meanwhile, the experimentation with LINDA brought into the light additional ways FILOS can be extended and adjusted to particular tuple-like process interaction approaches. Finally, an efficient message compression entails a better utilization of the network whilst delay, busy, and idle network periods decreases as the amount of the transported data quantities decreases; furthermore, the limited capacity of the underlying message queues "increases" in terms of queuing a larger number of smaller messages which improves asynchrony, whilst reliability increases since messages are broken into much fewer segments which decreases the possibilities of (Ethernet) collisions and transmission failures in general.

The research carried out in the previous chapters allowed the testing of ETHERLISP under pragmatic conditions. In chapter 4 algorithms encountering several representative problems were developed. The extended experimentation disclosed an expected weakening in performance for problems with very fine granularity, e.g. short execution times, whereas medium/coarse-grained algorithms performed very well. This behaviour depends partly on the system's structure and the additional processing power of the participating processors, and partly on the fashion algorithms are designed. In general, very fine granularities impose frequent communications that definitely are more expensive compared to references to a shared memory. Since ETHERLISP provides enough flexibility in designing alternative algorithmic approaches we found that divide-and-conquer is the most suitable method for extracting efficient parallelism from distributed systems; that is, data are partitioned and dispersed to multiple processes where an instance (or subinstance) of the algorithm handles them; evaluation results are collected and correlated for producing the final solution.

One of our major aims was the provision of a concurrent environment in which extended experimentation with various known or not parallel paradigms would be possible. In chapter 5 we develop and compare numerous popular models against ETHERLISP. From the point of view of implementors of ETHERLISP we observed that our system, theoretically in some cases and practically in some others, was superior compared with other network-based sys-

tems. As researchers novice in the field of concurrent processing we enlarged our knowledge about concurrency; we also felt able of considering and critisizing different approaches in extracting parallelism which led us to the conception and development of PRAXIS which is a complete model with novel semantics.

Finally, in chapter 6 we investigate the possibility of inventing new ways of expressing and extracting parallelism based on ETHERLISP. The outcome of this research was the PRAXIS paradigm. Its semantics are few, simple, novel, powerful and expressive enough to provide compact, elegant, and in some cases radical solutions to a broad range of problems. PRAXIS provides both synchronous and asynchronous approaches in applying parallelism with the former case as the most flavoured. The fundamental feature of the synchronous approach is the *praxis* message which consists of a particular condition applied immediately after the evaluation of an accompanied procedural entity; procedural entities are complete procedural blocks, usually inexpensive in CPU requirements, contain sequential ordinary code, and describe logically independent portions of an algorithm that can be scheduled for parallel execution. The loose dependency among *praxis* messages along with the embedded in them synchronization information yields an efficient and inexpensive scheduling of multiple interacting threads of control.

The final concluding remark of this research is that network-based systems are an interesting alternative to shared-memory architectures. The much less financial cost of a MIMD network configuration along with the high reliability provided from the current networking technology constitute an abstract and flexible environment in which effortless and highly efficient parallelism can be expressed and extracted.

Bibliography

[Akl89] Selim G. Akl *"The Design and Analysis of Parallel Algorithms"*, Prentice-Hall 1989.

[Andrews91] Gregory R. Andrews, *"Concurrent Programming Principles and Practice"*, The Benjamin/Cummings Publishing Company 1991.

[BABN84] Andrew D. Birrel and Bruce Jay Nelson, *"Implementing Remote Procedures Calls"*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp: 39-59.

[BRDD87] Robert G. Babb and David C. Dinucci, *"Design and Implementation of Parallel Programs with Large-Grain Data Flow"*, The MIT Press Scientific Computation Series, 1987.

[BKDR88] Brian W. Kernighan and Dennis M. Ritchie, *"The C Programming Language"*, Prentice-Hall, 1988.

[BKRP84] Brian W. Kernighan and Rob Pike, *"UNIX Programming Environment"*, Prentice-Hall, 1984.

[Brad93] Russell Bradford and Julian Padget, *"Concurrent Processing: Principles and Abstractions"*, Lecture notes for finall year course, Bath University, 1993.

[Boyle87] James Boyle et al., *"Portable Programs for Parallel Processors"*, Holt, Rinehart and Winston, 1987.

[Capo86] Capocelli R. M., Giancarlo R., and Taneja I. J., *"Bounds on the Redundancy of Huffman Codes"*, IEEE Trans. Inf. Theory 32, Nov. 6, 1986, pp:854-857.

[CBJM89] Christopher Burdorf and Jed Marti, *"Load Balancing Strategies for Time Warp on Multi-User Workstations"*, The RAND Corporation, Santa Monica, California, 1989.

[CDJ84] Fred B. Chambers, David A. Duce, and Gillian P. Jones, *"Distributed Computing"*, Academic Press 1984.

[Clamen89] S.M. Clamen, L.D. Leibengood, S.M. Nettles, and J.M. Wing, *"Reliable Distributed Computing with Avalon/Common Lisp"*, Carnegie Mellon University, CMU-CS-89-186 (September 1989).

[Comer88] Douglas Comer, *"Internetworking with TCP/IP Principles, Protocols, and Architecture"*, Prentice-Hall 1988.

[Cormack] Cormack G. V. and Horspool R. N., *"Algorithms for Adaptive Huffman Codes"*, Inf. Process. Lett. 18, Mar 3, 1984, pp:397-403.

[DHJF90] David Hutchinson and John Fitch, *"Implementing Timewarp using Lisp and Linda"*, Technical Report, University of Bath, 1990.

[Deitel84] Harvey M. Deitel, *"An Introduction to Operating Systems"*, Addison-Wesley, 1984.

[Dinh90] Nuong Quang Dinh, *"Timewarp and its Applications on a Distributed System"*, Ph.D Dissertation, University of Bath, 1990.

[Dollimore90] J. Dollimore, *"Programming Language Requirements for Distributed Systems"*, University of London, December 1990.

[EuLisp93] Padget J. and Nuyens G. (eds), *" The EuLisp Definition"*, University of Bath, Lisp and Symbolic Computation, 1993.

[Finkel87] Raphael A. Finkel, *"Large-Grain Parallelism: Three Case Studies"*, The MIT Press Scientific Computation Series, 1987.

[Flynn72] M. J. Flynn, *"Some Computer Organisations and their Effectiveness"*, IEEE Trans. Comput. C-21, 1972, pp:948-960.

[GAFS83] Gregory R. Andrews, Fred B. Schneider, *"Concepts and Notations for Concurrent Programming"*, Computing Surveys, Vol. 15, No. 1, March 1983, pp:3-41.

[GDBJ81b] Gelernter, D., and Bernstein, A.J. *"Distributed Communication via Global Buffer"*, Proc. Symp. on Principles of Distributed Computing, (August 1981) pp:10-18.

[Gehani90a] Gehani, N. H. *"Message Passing in Concurrent C: Synchronous versus asynchronous"*, Software-Practice and Experience 20, 6 (June 1990), pp:571-592.

[Gehani89b] Gehani, N. H. and Roome, W. D. *"The Concurrent C Programming Language"*, Silicon Press, Summit, NJ 1989.

[Gel85a] Gelernter, D. *"Generative Communication in Linda"*, ACM Trans. on Prog. Languages and Systems 7(1), (January 1985), pp:80-112.

[HAGJS85] Harold Abelson and Gerald Jay Sussman, *"Structure and Interpretation of Computer Programs"*, The MIT Press, 1985.

[Halst85] Robert H. Halstead, JR., *"Multilisp: A Language for Concurrent Symbolic Computation"*, ACM Trans. on Programming Languages and Systems, Vol. 7, No. 4, October 1985, pp:501-538.

[Hamilt84] Kenneth Graham Hamilton, *"A Remote Procedure Call System"*, Ph.D Disertation, Wolfson College, December 1984.

[Inmos84] Inmos Ltd., *"OCCAM Programming Manual"*, Prentice Hall, 1984.

[IRIS-4D87] IRIS-4D Series Manual, *"Communications"*, Silicon Graphics, Inc. 1987.

[Jef85] David R. Jefferson, *"Virtual Time"*, ACM TOPLAS, 7(3):404-425, 1985.

[JGD87] Editied by Leach H. Jamienson, Dennis B. Gannon, and Robert J. Douglass *"The Characteristics of Parallel Algorithms"*, Scientific Computation Series, The MIT Press 1987.

[JSJH80] J. F. Shoch and Jon A. Hupp, *"Measured Performance of an Ethernet Local Network"*, Comm. of the ACM, Vol. 23, No. 12, (December 1980), pp:711-721.

[KSSM75] Stephen R. Kimbleton and G. Michael Schneider, *"Computer Communication Networks: Approaches, Objectives, and Performance Considerations"*, Computing Surveys, Vol. 7, No. 3, September 1975.

[Kochan83] Stephen G. Kochan, *"Programming in C"*, Hayden Book Company, 1983.

[Leich89] Jerrold Sol Leichter, *"Shared Tuple Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity"*, PHD Disertation, Yale Univ., YALEU/DCS/TR-714, July 1989.

[Leffler89] J. Lefler et al, *"The Design and Implementation of the 4.3BSD UNIX Operating System"*, Addison-Wesley, 1989.

[Lelewer87] Debra A. Lelewer and Daniel S. Hirschberg, *"Data Compression"*, ACM Computing Surveys, Vol. 19, No. 3, September 1987, pp: 261-295.

[Liskov79a] Liskov, B. *"Primitives for Distributed Computing"*, Proceedings of the Seventh ACM Symposium on Operating System Principles, pp:33-42, Pacific Grove, California (December 1979).

[Liskov82b] Liskov, B. *"On Linguistic Support for Distributed Programs"*, IEEE Transactions on Software Engineering SE-8(3), pp:203-210 (May 1982).

[MacLach92] Robert A. MacLachlan (editor), *"CMU Common Lisp User's Manual"*, Carnegie Mellon University Technical Report CMU-CS-92-161, July 1992.

[NCDG86] Nicholas Carriero and David Gelernter, *"The S/Net's Linda Kernel"*, ACM Trans. on Computer Systems, Vol. 4, No. 2, May 1986, pp:110-129.

[NCDG89] Nicholas Carriero and David Gelernter, *"Linda in Context"*, Commun. of the ACM, Vol. 32, No. 4, April 1989, pp:444-458.

[NCDGJL86] Nicholas Carriero, David Gelernter and Jerry Leichter, *"Distributed Data Structures in Linda"*, In proceedings of the ACM Symposium on Principles of Programming Languages (St. Petersburg, Fla., Jan 13-15, 1986).

[Needham79] R. Needham, *"System Aspects of the Cambridge Ring"*, ACM Seventh Symp. on Operating Systems Principles, pacific Grove, California (December 1979).

[Nelson81] Nelson, B. J. *"Remote Procedure Call"*, Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, Calif. 1981.

[Nichols90] David A. Nichols, *"Multiprocessing in a Network of Workstations"*, Ph.D Dissertation, Carnegie Mellon University, 1990.

[PJFJ87] J. A. Padget and J. P. Fitch, *"Concurrent Object-oriented Programming"*, Bath Computer Science, Technical Report 87-05, 1987.

[PTMK88] Pete Tinker and Morry Katz, *"Parallel Execution of Sequential Scheme"*, ACM, 1988.

[Penny82] B. K. Penny and A. A. Baghdadi, *"Survey of Computer Communications Networks"*, Research Report 78/42, Imperial College of Science and Technology, 1982.

[Ring88] Ringwood, G.A., *"Parlog86 and the Dining Logicians"*, Commun. ACM 31, 1 (Jan. 1988), pp: 10-25.

[SKPW89] Stephen G. Kochan and Patrick H. Wood Consulting Editors, *"Unix Networking"*, Pipeline Associates, 1989.

[Spector82] Spector, A. Z. *"Performing Remote Operations Efficiently on a Local Computer Network"*, Commun. ACM 25, 4 (April 1982), 246-260.

[Steele90] Guy L. Steele JR., *"Common LISP The Language"*, Digital Equipment Corp. (second edition), 1990.

[Sunder90] V. S. Sunderam, *"PVM: A Framework for Parallel Distributed Computing"*, Emory University, Atlanda, 1990.

[SHGS87] Samuel P. Harbison and Guy L. Steele JR., *"C: A Reference Manual"*, Tartan Laboratories, Prentice-Hall, 1987.

[SunOS] *"Network Programming"*, Sun Microsystems, Vol. 10.

[UDNMcD90] Ulf Dahlen and Neil MacDonald, *"Scheme-Linda"*, In proceedings of EUROPAL workshop on High Performance and Parallel Computing in Lisp, November 1990.

[White88] Robert A. Whiteside and Jerrold S. Leichter, *"Using Linda for Supercomputing on a Local Area Network"*, In Proceedings, Supercomputing '88, 1988.

[XEROX81] *"COURIER: The Remote Procedure Call Protocol"*, Xerox System Intergration Standard XSIS-038112, Xerox Corporation, Stamford, Connecticut, Dec. 1981.

[Yuen90] C. K. Yuen, *"Quicksort in BaLinda Lisp: A Study in Language Extensions to Meet Algorithm Requirements"*, DISCS NUS, Kent Ridge, Singapore 0511, 1990.

[Zayas87] Edward R. Zayas, *"The Use of Copy-On-Reference in a Process Migration System"*, Ph.D Dissertation, Carnegie Mellon University, 1987.

Appendix A

EtherLISP User's Manual

A.1 Introduction

IN THIS APPENDIX we describe all operations provided by the ETHER kernel's primitives. The operations are fall into four general categories: (a) the *fundamental primitives* for developing distributed applications, (b) *message manipulation primitives* for constructing messages, and (c) *general purpose primitives* used for simplifying the process of designing and implementing efficient applications. The primitives are also categorized as *generic* or *ordinary*; the first category includes those primitives whose execution is allowed at any endpoint of a channel, whereas ordinary primitives are applied in the address space of the initiating process (root) only. A fairly large number of primitives could be implemented on user's behalf by using the fundamental operations; but we believe that their implementation in the C language yields their faster execution and hence, a better overall performance of ETHERLISP. However, the number and activities of the primitives have been currently proved adequate for developing integrated and efficient applications as we showed with the PRAXIS model in chapter 6.

A.2 The Fundamental Primitives

• (**make-ether** *host* &key *(service 'packet)*) ⟶ *ether object* [Generic Function]

Creates a bidirectional communication channel whose remote endpoint on host ***host*** is a complete Lisp evaluator. The returned value is a fresh first class object of type **ETHER** which names the newly created remote thread of control (RTC). The communication service can be of the type indicated by `service`; the keyword *stream* employs the TCP/IP communications protocol whilst the (default) keyword *packet* utilizes the connectionless UDP/IP service. The returned object is characterized as permanently-bound object since any accidental or not assignment of a new value will cause the loss of the connected process.

• (**etherp** *eth*) ⟶ { *t* | *nil* } [Generic Function]

The type of an ETHER object is verified either by `etherp()` that returns `t` or `nil`, or by the generic `type-of()` Lisp primitive.

• (**push-ether** *expr* &optional *eth*) ⟶ *t* [Generic Function]

Messages can be sent in both directions of a communication channel. If the sender is the root (LTC) then *expr* is directed to the RTC specified by *eth*; when the caller performs remotely *eth* is omited since there is no need for multiplexing messages at any distant location. The absence of the receiver signals an error when reliable service, i.e. TCP/IP, is in use. An error is also signalled when *expr* encapsulates a non-transmittable object type.

• (**ether-readable-p** &optional *eth*) ⟶ { *t* | *nil* } [Generic Function]

Some concurrent languages provide a non-blocking receive operation. In *Concurrent C* this is provided by the *select* statement that waits for the arrival of the first transaction call. Precisely, select polls on a collection of *guards*, which are boolean expressions, and they are postfixed by *alternatives*. When a guard eventually evaluates to true, i.e. a transaction call has been completed, the corresponding alternative is evaluated. In ETHER the analogous primitive examines the existence or not of any pending messages in the queue of an *eth*, and returns `t` or `nil` respectively.

• (**listen-ether** &optional *eth*) ⟶ *any* [Generic Function]

The receipt of a message is a clear way to synchronize distributed processes, since it behaves similarly to the *wait* semaphore operation. The receiver blocks (waits) until a caller sends (signals) a message. The primitive's asynchronous semantics derive by the assignment of a message buffer upon creation of a receiver; hence, any messages sent are accepted and queued even if the receiver executes code due to previously accepted messages. Assuming that the queue space is enough, messages can be received and evaluated at a later time in

a *first-in first-out (FIFO)* order. Again, the parameter *eth* is omited in case of a remote recipient.

• **(select-ether** &key *(block t)*) ⟶ *nil* | *ether object* [Function]

Execution of multiple concurrent processes gives rise to the need for scheduling and synchronization. In concurrent programs processes must interact in mutual exclusion ensuring that critical sections are not (erroneously) accessed simultaneously. Moreover, proper scheduling entails the optimal utilization of the system's resources. The build-in tool for the accomplishment of these purposes is the `select-ether()` operation that picks up and returns a readable (with pending messages) *ether*. Selection blocks when there is not any readable channel unless the keyword *:block* is set to `nil`. Selection can be requested according to several channel properties such as message traffic (ether load), or scheduling schemes such as the round robin policy.

• **(kill-ether** *&rest eths*) ⟶ *nil* [Function]

An explicit process termination mechanism is provided when remote evaluation is no longer required. After a process has been killed, the process enters in a *close* state and further communication is not allowed. A warning message is signaled when a process has pending messages upon killing it.

A.3 Special Variables

• ***number-of-ethers*** ⟶ { *0* | *number* } [Special Variable]

Indicates the number of active RTC's.

• ***ether-list*** ⟶ { *list* | *nil* } [Special Variable]

Maintains all active RTC's as anonymous channel entities; thus, any operation requiring the name of an RTC among its arguments can be performed via the *i*th element of this list.

• ***remote*** ⟶ { *t* | *nil* } [Generic Special Variable]

A communication channel in ETHERLISP has the local and the remote endpoints. It is essential to distinguish between endpoints since they might connect two processes executing in different address spaces as usually is the case. This special variable returns `nil` at the local endpoint and `t` at the remote one. This information is of great importance because it allows the development of generic, compact, and efficient code; for example, the body of ETHERLISP's sending primitive acts according to the environment in which it is applied (see the description of the `push-ether()` primitive).

• ***last-smsg*** ⟶ *any* [Special Variable]

- ***last-rmsg*** ⟶ *any* [Special Variable]
- ***last-smsg-size*** ⟶ *byte size* [Special Variable]
- ***last-smsg-size*** ⟶ *byte size* [Special Variable]

These special variables hold the last send or received message, as well as their size in bytes. They are mainly used for debugging or for a rough estimation of the communication cost.

- ***display*** [Special Variable]

Apart from the trivial errors such as the possibility of partial failure (host crash), garbled, duplicated, or lost messages, logical errors are very often when developing distributed applications. It is of common practice the insertion of temporary comments printed ar critical portions of code at run-time. A problem arises when numerous comments corresponding to individual processes have to be distinguished; moreover further distinction of the endpoints of a communication channel an action takes place is needed. Thus, the additional code for debugging purposes grows substantially and the actual code becomes complicated and obscure. To overcome this problem we introduce the special variable `*display*` that when evaluates to `t` forces each process running on a separate visual window and the automatic display of both received messages and results produced after evaluation. In this way logical errors are detected at a glance.

A.4 Message Manipulation Primitives

- (**neval** *expr*) ⟶ *expr* [Generic Special Form]

Its semantics are identical to `identity()` except that no evaluation is performed on the supplied argument *expr*. The returned value is *expr* itself. It attends upon the need of transmitting symbol print names instead of their values in case the value is known and intact in the receiving scope. It is also needed when evaluation is not desired in the sender's scope.

- (**qeval** *expr*) ⟶ *quoted any* [Generic Function]

Returns the quoted result after *expr* has been evaluated. It is applied for messages whose first component is a function name with arguments data lists.

- (**make-msg** &rest *exprs*) ⟶ *any* [Generic Function]

Compound messages are constructed by this primitive. The message's components ***exprs*** are evaluated in order and the evaluation results constitute the successive elements of a message formulated as a list.

A.5 General Purpose Primitives

• (**ether-id** *eth*) ⟶ *identification number* [Generic Function]

Often it is convenient to distinguish multiple communication channels by identification numbers instead of their symbolic names. This primitive returns a small system-widely unique integer (id) which identifies an RTC.

• (**ether-type** *eth*) ⟶ { *stream* | *packet* } [Generic Function]

Returns a symbol indicating the type of the channel *eth*. A return value *stream* denotes a TCP/IP connection scheme whereas the value *packet* indicates the use of the UDP/IP protocol.

• (**ether-state** *eth*) ⟶ { *connected* | *closed* } [Generic Function]

Returns a symbol indicating the current state of the channel *eth*. A channel has two states: a *connected* and a *closed* one.

• (**ether-hosts** *eth*) ⟶ *cons* [Function]

Returns a cons whose `car` and `cdr` indicate the names of the hosts of the endpoints of the channel *eth*.

• (**ether-load** *eth*) ⟶ *any* [Function]

Informs the current activity of an RTC in terms of returning the total number of messages that have been bidirectionally transported via the channel *eth*. Useful as a work-load scheduling criterion.

• (**verify-host** *host-name*) ⟶ { *t* | *nil* } [Generic Function]

Host names are maintained in the *hosts* database which may be contained in the */etc/hosts* file, the Network Information Service (NIS) hosts database, the Internet domain name server, or a combination of these. In particular, each host has one official name, the first name in the database entry, and optionally one or more nicknames. Either official names or nicknames may be specified from the argument *host-name*. This primitive returns `t` is either a valid host official name or nickname; `nil` otherwise.

• (**init-ether** *fname* &rest *eths*) ⟶ *t* [Function]

A file *fname* which contains the application being executed is loaded and evaluated simultaneously by all connected RTC's. Returns `t` on success; otherwise an appropriate error message.

• (**reval** *expr* &rest *eths*) ⟶ *any* [Function]

Distributed processes interact by exchanging messages. In the simplest case, this involves two processes; one process, called the *caller* or *client*, initiates the interaction, while the

other process, called the *receiver* or *server*, waits (blocks) for the interaction. At a later time processes may exchange roles. Such an interaction is called a *transaction call.* There are two main types of transaction calls; *synchronous*, where the caller sends messages and immediately waits for the receiver to acknowledge the messages' acceptance; the receiver evaluates messages and returns some result(s) to the caller. At this point, the caller can resume execution. Obviously, a synchronous transaction call implies both a strict synchronization, and a bidirectional communication channel between the interacting processes. Even in case that the receiver does not return any result, i.e. a call for synchronization purposes, the caller still waits the receiver to complete the call. Conversely, an *asynchronous* transaction call the caller after has sent a message, immediately resumes execution; that is, no synchronization is required since the caller does not wait from the receiver neither any acknowledgement nor result. Clearly, messages are exchanged in a unidirectional manner. Intergrated transaction calls are provided by the primitive `reval()`. Any expression *expr* is transmitted to an arbitrary number of RTC's whereupon evaluation results serve as acknowledgements. The value returned is the last result received.

• (**listen-ether-within** *dur eth err*) ⟶ *any* | *err* [Generic Special Form]

Timed receive operations provide an asynchronous alternative way of receiving messages. The receiver immediately resumes execution either when a message has been arrived, or when the time interval *dur* expires. The later causes the evaluation of an expression specified by *err*; this is essential because, for example, there is no way to distinguish between a `nil` returned due to a timed out operation or a `nil` returned as the result of the last remote evaluation. However, *err* is a user-dependent expression.

• (**broadcast-ether** *expr*) ⟶ *t* [Function]

Support for a broadcast service is also provided. All active processes, including root, receive and evaluate *expr* but the result is remotely swallowed; that is, remote processes after the evaluation immediately switch to the next receive operation without returning any result, except when the message causes an error.

• (**make-shadow-ether** *host* &optional *service*)⟶ *nil* [Function]

Creates an evaluator on host *host* of service type *service*. Unlike `make-ether()` this primitive places its output on the `*ether-list*` instead of assigning it to a user-supplied identifier. Shadowed evaluators are anonymous whilst access to them is allowed in terms of referring to the *i*th element of `*ether-list*`

• **(peek-ether** *eth*) ⟶ *any* [Generic Function]

The last alternative way of receiving messages is by peeking them. That is, a peeked message is treated as still unread, since successive peek(s) return the previously peeked message. The message is removed (read) from the queue when a receive is performed. Clearly, peek blocks if there is no pending messages. Peeking messages is very important because a user multireceive process can avoid queuing prematurely sent messages, by means of lack of resource availability, since these messages have been already queued by the underlying manager. Consequently, their evaluation can be postponed for a little later while other channels are examined. The later implies an optimal scheduling of the system resources as in the case of a disk controller in a distributed operating system.

• **(ping-ether** *eth dur*) ⟶ { *t* | *nil* } [Generic Function]

Repeatedly sends a message to an RTC and reports whether or not a reply was received by returning `t` or `nil` respectively. It keeps trying until *dur* time interval has expire or an answer is received.

• **(flush-ether** &rest *eths*) ⟶ *nil* [Function]

Transparently withdraws message(s), if any, pending in the queues of the supplied channels *eths* and returns `nil`. This operation is useful when unrecoverable error are signalled during execution of an application; thus, "clean" channels should be provided for a correct restart of the application.

• **(rollback-ether** &rest *eths*) ⟶ *nil* [Function]

When deadlock is detected the system must be able to stop it happening and to recover when it has happened. It is clear that recovering from deadlock is not so simple due to the nature of the *state* of a concurrent program, which can be thought as the values of the program's variables at some point in time. That is, a program starts execution in some state whilst individual processes execute at their own rates by altering the global state of the program. Clearly, the problem becomes more complicated when processes share a network. Since deadlock recovery is not included in the current goals of our research, the primitive `ether-rollback()` provides a naive recovery facility attempting to bring the (erroneous) program's state back to its initial one. Precisely, the following actions are performed; (a) all processes are interrupted, (b) all message queues are flushed, and (c) all communication channels are checked if they are alive. The value returned is either `t` or `nil` denoting the success or not of the rollback. In the former case, the application program may be started again.

Appendix B

The Dining Philosophers Solved in the Socket Package

B.1 Defining the Problem

THE PROBLEM OF THE DINING PHILOSOPHERS [Ring88, NCDG89, Andrews91] is defined as following: Five philosophers spend their lives eating spaghetti and thinking. They eat at a circular table in a dining room. The table has five chairs around it and a chair has been assigned to each philosopher. There are also five chopsticks on the table and each philosopher must have in his possession two chopsticks in order to eat. If a philosopher can not get (grab) two chopsticks at once then he must wait until he will be able to get them. The chopsticks are picked up one at a time. When a philosopher has finished eating he puts (releases) the chopsticks on the table and leaves the refectory.

The interesting point in this example is an endless competition for mutually exclusive access to more than one resource among several processes. Since each philosopher acts independently of the others the problem can be characterized as asynchronous in nature.

B.2 Solving the Problem

The main reasons of re-mentioning this problem are the extended presentation of the SOCKET package (section 5.7) by using an illustrative example, and the comparison of the size of this solution with the corresponding PRAXIS's one (section 6.3.6). Furthermore, we intend to show some of the attributes of ETHERLISP which came as the result of our prior experimentation with the Unix socket constructs.

The solution presented in figure B-1 employs the Connectionless Communication Service (CCS), i.e. the UDP/IP protocol, and the Socket Address Service (SAS) of the SOCKET package. The solution presented requires seven processors each assigned to one philosopher object (remote process), whilst the central handler and SAS are assigned to a totally devoted processor. Interconnections to sockets are performed through the transparent mechanism of SAS which passively listens on the well-known IP address [1] held by `*sas-addr*`. After a connection to SAS has been established the handler starts all processes and propagates their IP addresses via SAS. Note that the sockets are first created on the proper network locations and then messages are multiplexed/demultiplexed to/from them via user-defined system-widely unique names. This mechanism triggered the necessity of the RTC mapper `*ether-list*` which handles communication channels as anonymous elements of a list. Finally, the handler is responsible for starting, serving, and terminating all remote processes including SAS. For maximum efficiency any unsatisfied request either for an unavailable ticket or chopstick is appended to the corresponding waiting queue. However, this burden due to queue management is absent from PRAXIS since the scheduler removes messages from the queues associated with channels only when the conditions enclosed in *praxis* messages are satisfied.

[1] Since we have not officially reserved a specific permanent IP address for SAS we arbitrarily declare one.

```
(defvar *phil-num* 5)                    ; Traditionally five philosophers.
(defvar *sas-addr* '("lumina" . 2586)) ; SAS well-known IP address.
(defvar *dpdir* "dinphil.lsp"            ; The file containing the application.
(defvar *time-out* 30)                   ; Time out interval (30 seconds).
(defvar *init-msg* t)                    ; Initialization message id.
(defvar *stop-msg* t)                    ; Termination message id.
(defvar *rounds* 50)                     ; Repeat simulation 50 times.

(defvar *phil-in-sock-name-table* (make-array *phil-num*
         :initial-contents '(pin0 pin1 pin2 pin3 pin4)))      ; Input socket names.
(defvar *phil-in-sock-table* *phil-in-sock-name-table*)       ; Input sockets.
(defvar *phil-out-sock-name-table* (make-array *phil-num*
         :initial-contents '(pout0 pout1 pout2 pout3 pout4))) ; Output socket names.
(defvar *phil-out-sock-table* *phil-out-sock-name-table*)     ; Output sockets.
(defvar *handler-in-sock-name*  'hin)                         ; Handler input socket name.
(defvar *handler-in-sock* *handler-in-sock-name*)             ; Handler input sockets.
(defvar *handler-out-sock-table* (make-array *phil-num*
         :initial-contents '(hout0 hout1 hout2 hout3 hout4)));Handler output socket names.

(defvar *phil-count* 0)
(defvar *phil-num-in-room* (- *phil-num* 1)) ; Only four philosophers in the refectory.
(defvar *stop-count* 0)
(defvar *ticket* -4)       ; Ticket request.
(defvar *fork* -3)         ; Left or right hand chopstick request.
(defvar *wait* -2)         ; Request ticket or chopstick.
(defvar *signal* -1)       ; Release ticket or chopstick.
(defvar *ack* t)           ; Acknowledgement message.
(defvar *fork-free* 1)
(defvar *fork-occupied* 0)
(defvar *fork-queue* nil)  ; Chopstick waiting queue.
(defvar *room-queue* nil)  ; Ticket waiting queue.
; All chopsticks are initially free.
(defvar *fork-table* (make-array *phil-num* :initial-contents '(1 1 1 1 1)))

(defun philosopher (phil)  ; A philosopher simulation object.
  (let ((left-fork phil) (right-fork (mod (+ phil 1) *phil-num*)))
    (dotimes (i *rounds*)
      (format t "~%ROUND: ~D" i)
      (format t "~%Philosopher ~D is thinking..." phil)
      (wait phil *ticket*)
      (wait phil left-fork)
      (wait phil right-fork)
      (format t "~%Philosopher ~D is eating..." phil)
      (signal phil left-fork)
      (signal phil right-fork)
      (signal phil *ticket*))))

(defun wait (phil x) ; Like the blocking P semaphore operation.
  (throw-message (aref *phil-out-sock-table* phil)
    (cons phil (cons *wait* x)))
  (catch-message (aref *phil-in-sock-table* phil)))

(defun signal (phil x) ; Like the signaling V semaphore operation.
  (throw-message (aref *phil-out-sock-table* phil)
    (cons phil (cons *signal* x))))
```

TO BE CONTINUED

```
(defun get-ticket-p (x) (if (= x *ticket*) t nil))
(defun enter-room-p () (if (< *phil-count* *phil-num-in-room*) t nil))
(defun leave-room-p (x) (if (= x *ticket*) t nil))
(defun get-fork-p (fork) (if (= (aref *fork-table* fork) *fork-free*) t nil))

(defun handler () ; Input/Output message handler.
  (prog ((counter 0) (msg) (phil) (msg-type) (fork) (sem) (first-phil))
READ-MSG
    (setq msg (catch-message *handler-in-sock*)) ; Read next message.
    (when (equal msg *stop-msg*)                 ; Check if simulation has been finished.
          (incf *stop-count*)
          (when (and (= *stop-count* *phil-num*)
                     (not (catcher-readable-p *handler-in-sock*))) (return))
          (go READ-MSG))
    (setq phil (car msg))        ; Decompose message.
    (setq msg-type (cadr msg))
    (setq fork (cddr msg))
    (setq sem fork)
    (format t "~%< ROUND: ~4D > Phil [~D] " (incf counter) phil)
    (when (= msg-type *wait*)        ; If message is a request then
          (when (get-ticket-p sem)  ; if it is ticket request
                (if (enter-room-p)  ; check if no more that four philosophers are in.
                    (progn (format t "gets ticket.") ; Grand a ticket.
                           (incf *phil-count*)
                           (throw-message (aref *handler-out-sock-table* phil) *ack*)
                           (go READ-MSG))
                    (progn (format t "waits for ticket.") ; Block the philosopher.
                           (setq *room-queue* (append *room-queue* (list phil)))
                (go READ-MSG)))) ; Read next message.
      (if (get-fork-p fork)        ; If message is a chopstick request then
          (progn (if (= phil fork) ; determine if it is a left or right hand
                     (format t "gets left fork.")
                     (format t "gets right fork. Starts EATING..."))
                 (setf (aref *fork-table* phil) *fork-occupied*)
                 (throw-message (aref *handler-out-sock-table* phil) *ack*)
                 (go READ-MSG)) ; Inform remote philosopher to start eating.
          (progn (when (= phil fork) ; Block the philosopher and insert him in the queue.
                       (format t "waits for left fork.")
                       (format t "waits for right fork."))
                 (setq *fork-queue* (append *fork-queue* (list phil)))
          (go READ-MSG)))) ; Read next message.
    (when (= msg-type *signal*) ; If message is a realising ticket request then
      (when (leave-room-p sem)  ; free a ticket.
            (format t "leaves room. Starts THINKING... ")
            (decf *phil-count*)
            (when (and *room-queue* (enter-room-p)) ; Check ticket queue first.
                  (incf *phil-count*)
                  (setq first-phil (pop *room-queue*)) ; Grand a queued philosopher.
                  (format t "Phil ~D takes his place." first-phil)
                  (throw-message (aref *handler-out-sock-table* first-phil) *ack*))
            (go READ-MSG))
      (if (= phil fork) ; If message is a releasing chopstick request then
          (format t "leaves left fork. ")
          (format t "leaves right fork. "))
      (setf (aref *fork-table* fork) *fork-free*) ; free a chopstick.
      (when *fork-queue* ; Check chopstick queue first.
            (setq first-phil (pop *fork-queue*)) ; Grand a queued philosopher.
            (format t "Phil ~D gets it." first-phil)
            (throw-message (aref *handler-out-sock-table* first-phil) *ack*))
      (go READ-MSG)))) ; Read next message.
```

TO BE CONTINUED

```
(defun init-phil-in-link (phil) ; Propagate via SAS all input socket IP addresses.
  (let ((link (aref *phil-in-sock-name-table* phil)))
    (setf (aref *phil-in-sock-table* phil) (make-catcher))
    (propagate-address *sas-addr* link (aref *phil-in-sock-table* phil))))

(defun init-phil-out-link (phil) ; Connect remote process via SAS to the central handler.
  (setf (aref *phil-out-sock-table* phil)
        (connect-thrower (get-address *sas-addr* *handler-in-sock-name*))))

(defun send-init-msg-to-handler (phil) ; Inform the handler for a successful connection.
  (throw-message (aref *phil-out-sock-table* phil) *init-msg*))

(defun wait-to-start-phil (phil)
  (send-init-msg-to-handler phil)
  (format t "~%*** Waiting for initial message...~%")
  (catch-message (aref *phil-in-sock-table* phil))
  (philosopher phil)
  (terminate-phil phil))

(defun terminate-phil (phil) ; Shutdown all input and output sockets.
  (format t "~2%Philosopher ~D is terminated." phil)
  (throw-message (aref *phil-out-sock-table* phil) *stop-msg*)
  (catch-message (aref *phil-in-sock-table* phil))
  (close-catcher (aref *phil-in-sock-table* phil))
  (close-thrower (aref *phil-out-sock-table* phil)) (by))

(defun init-philosopher (phil) ; Start a philosopher simulation object.
  (init-phil-in-link phil)
  (init-phil-out-link phil)
  (wait-to-start-phil phil))

(defun init-handler-in-link () ; Make known via SAS the handler's IP address.
  (setq *handler-in-sock* (make-catcher))
  (propagate-address *sas-addr* *handler-in-sock-name* *handler-in-sock*))

(defun init-handler-out-links () ; Connect handler via SAS to all remote IP addresses.
  (prog ((i 0))
  LOOP
    (when (= i *phil-num*) (return))
    (setf (aref *handler-out-sock-table* i)
      (connect-thrower (get-address *sas-addr* (aref *phil-in-sock-name-table* i))))
    (incf i)
    (go LOOP)))

(defun start-all-philosophers ()
  (let ((addr0 (get-address *sas-addr* (aref *phil-in-sock-name-table* 0)))
        (addr1 (get-address *sas-addr* (aref *phil-in-sock-name-table* 1)))
        (addr2 (get-address *sas-addr* (aref *phil-in-sock-name-table* 2)))
        (addr3 (get-address *sas-addr* (aref *phil-in-sock-name-table* 3)))
        (addr4 (get-address *sas-addr* (aref *phil-in-sock-name-table* 4))))
    (broadcast-message *init-msg* addr0 addr1 addr2 addr3 addr4)))

(defun terminate-all-philosophers ()
  (let ((addr0 (get-address *sas-addr* (aref *phil-in-sock-name-table* 0)))
        (addr1 (get-address *sas-addr* (aref *phil-in-sock-name-table* 1)))
        (addr2 (get-address *sas-addr* (aref *phil-in-sock-name-table* 2)))
        (addr3 (get-address *sas-addr* (aref *phil-in-sock-name-table* 3)))
        (addr4 (get-address *sas-addr* (aref *phil-in-sock-name-table* 4))))
    (broadcast-message *stop-msg* addr0 addr1 addr2 addr3 addr4)))
```

TO BE CONTINUED

```
(defun terminate-handler () ; Terminate simulation.
  (format t "~2%*** DINING PHILOSOPHER COMPLETED ***~2%")
  (close-sas *sas-addr*)
  (close-catcher *handler-in-sock*)
  (dotimes (i *phil-num*) (close-thrower (aref *handler-out-sock-table* i))))

(defun check-all-philosophers () ; Ensure that all connections are alive.
  (dotimes (i *phil-num*)
    (catch-message *handler-in-sock*)
    (format t "~%Philosopher ~D is waiting to start..." i))
  (format t "~2%*** DINING PHILOSOPHERS START ***~2%"))

(defun distribute-philosophers () ; Start philosopher objects on explicitly selected hosts.
  (remote-akcl "lumina" *dpdir* (make-socket-address-mapper 2586))
  (format t "~%Network socket address service SAS allocated...")

  (when (not (ping-sas *sas-addr* *time-out*))
        (format t "~%Cannot make socket address mapper on ("lumina" . 2586)") (by))
  (init-handler-in-link)
  ; "lumina", "borodin" etc are host names.
  (format t "~%Init handler in link established...")
  (remote-akcl "lumina" *dpdir* (progn (load *dpdir*) (init-philosopher 0)))
  (format t "~%Philosopher 0, allocated...")
  (remote-akcl "borodin" *dpdir* (progn (load *dpdir*) (init-philosopher 1)))
  (format t "~%Philosopher 1, allocated...")
  (remote-akcl "coign" *dpdir* (progn (load *dpdir*) (init-philosopher 2)))
  (format t "~%Philosopher 2, allocated...")
  (remote-akcl "dingle" *dpdir* (progn (load *dpdir*) (init-philosopher 3)))
  (format t "~%Philosopher 3, allocated...")
  (remote-akcl "cubby"  *dpdir* (progn (load *dpdir*) (init-philosopher 4)))
  (format t "~%Philosopher 4, allocated..."))

(defun init-handler () ; Start simulation.
  (distribute-philosophers)
  (check-all-philosophers)
  (init-handler-out-links)
  (start-all-philosophers)
  (handler)
  (terminate-all-philosophers)
  (terminate-handler))
```

Figure B-1: The dining philosophers solved in the SOCKET package.

Appendix C

EtherC: A Concurrent Approach to the ANSI C Language

IN THIS APPENDIX we present an interface that provides concurrent capabilities to the ANSI C language [BKDR88, Kochan83]. This interface actually follows the same patterns as the ones of the SOCKET package presented in section 5.7. The only difference lays in the syntax of the applied primitives. However, our aim is only to illustrate the very basic application of the socket abstraction [SunOS, Vol:10] in a non-Lisp environment.

The application consists of two independent programs: a passive server (figure C-1) that continuously echoes any receive message formulated as a sequence of characters; echo requests are issued by an active client (figure C-2) echo requests are issued by an active client which performs an endless round of sending any message read from the standard input until a return is pressed. Assuming that the server and the client are named as **c_server** and **c_client** then the server is started by typing

Host-1%**c_server**

whilst the client is activated by typing

Host-2%**c_client** $< Host - 1 >$ $< port - number >$

where *port-number* is an arbitrarily selected port on host *Host-1* displayed by the server. The first message interchanged is called a *handshaking* where the server is informed the destination address. The astute reader notices from the code that the client is alternatively interrupted when Control-Q is pressed in which case the remote server is also terminates since it receives an end-of-session (EOS) message.

```
#include "sockets.h"
main() {
    LISTENER listener;          /*  listener structure  */
    LISTENERID listenerid;      /*  listener IP address structure  */
    SOCKET in_sock, out_sock;   /*  input and output socket structures  */
    char buf[100], c_name[20];  /*  communications buffer of size 100 bytes  */
    int c_port;                 /*  TCP/IP 32-bits port number  */

    listener = make_listener();         /*  create a listener  */
    listenerid = listener_id(listener); /*  get its IP address (hostname and port)  */
    printf("echo server: listening on #<%s %d>\n", listenerid->h_name, listenerid->h_port);

    in_sock = listen_on(listener);          /*  accept foreign connections  */

    strcpy(c_name, socket_read(in_sock));  /*  receive sender's IP address  */
    strcpy(buf, socket_read(in_sock));
    stoi(buf, &c_port); /*  convert char to integer  */

    out_sock = connect_to(c_name, c_port); /*  connect to the foreign sender  */
    socket_write(out_sock, "echo server ok...");

    while(1) {
        strcpy(buf, socket_read(in_sock)); /*  read  a message  */
        if (strcmp(buf, "EOS") == 0) {
            printf("*** eof ***\n");
            socket_write(out_sock, "EOS");
            break;
        }
        printf("%s\n", buf);
        socket_write(out_sock, buf);        /*  echo a message  */
    }
    close_listener(listener);               /*  close all sockets  */
    close_socket(in_sock);
    close_socket(out_sock);
    printf("echo server: exiting...\n");
    exit(0);
}
```

Figure C-1: Th echo server.

```
#include "sockets.h";
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

void on_quit()
{
    longjmp(begin, 0);
}

main(argc, argv) int argc; char **argv; {
    LISTENER listener;          /*  listener structure  */
    LISTENERID listenerid;      /*  listener's IP address structure  */
    SOCKET in_sock, out_sock;   /*  input and output socket structures  */
    char buf[100];              /*  communications buffer of size 100 bytes  */
    int rcnt;                   /*  count network-read bytes  */
    void on_quit();

    listener = make_listener();
    listenerid = listener_id(listener);
    printf("#<client %s %d>\n", listenerid->h_name, listenerid->h_port);

    out_sock = connect_to(argv[1], atoi(argv[2])); /*  connect to server  */
    socket_write(out_sock, listenerid->h_name);    /*  send client's address  */
    itos(listenerid->h_port, buf); /*  convert integer to string  */
    socket_write(out_sock, buf);

    in_sock = listen_on(listener); /*  wait for server to confirm connection  */
    strcpy(buf, socket_read(in_sock));
    printf("==>%s\n", buf);

    while(1) { /* keep sending messages until Ctr-C is pressed  */
        if (setjmp(begin) == 0) {
            signal(SIGQUIT, on_quit);
            if ((rcnt = read(0, buf, sizeof(buf))) <= 0) {
                if (rcnt == 0) {
                    socket_write(out_sock, "EOS");
                    break;
                } else if (rcnt < 0) {
                    fprintf(stderr, "*** Error: read failed\n");
                    socket_write(out_sock, "EOS");
                    break;
                }
            }
            buf[rcnt] = '\0';
            socket_write(out_sock, buf);        /*  send message  */
            strcpy(buf, socket_read(in_sock)); /*  read ''echoed'' message  */
            printf("==>%s\n", buf);
        } else {
            printf("Quit...\n");
            socket_write(out_sock, "EOS");      /*  terminate remote server  */
            break;
        }
    }
    close_listener(listener);                   /*  close all sockets  */
    close_socket(in_sock); close_socket(out_sock); exit(0);
}
```

Figure C-2: The echo client.